CHOReOS Middleware Specification (D3.1)

Apostolos Zarras, Nikolaos Georgantas, Thiago Teixeira, Sara Hachem, Valérie Issarny, Fabio Kon, Pierre Châtel, Amira Ben Amida, Santos Carlos Eduardo Moreira Dos, Rafael Correia, et al.

CHOReOS

Large Scale Choreographies for the Future Internet

ICT IP Project

Deliverable D3.1

CHOReOS Middleware Specification

http://www.choreos.eu

SEVENTH FRAMEWORK PROGRAMME

Project Number	:	FP7-257178
Project Title	:	CHOReOS Large Scale Choreographies for the Future Internet

Deliverable Number	:	D3.1
Title of Deliverable	:	CHOReOS Middleware Specification
Nature of Deliverable	:	Report
Dissemination level	:	Public
Licence	:	Creative Commons Attribution 3.0 License
Version	:	VA.0
Contractual Delivery Date	:	1/10/2011
Contributing WP	:	WP3
Author(s)	:	Apostolos Zarras (UOI), Nikolaos Georgantas (INRIA), Thiago Teixeira (USP), Sara Hachem (INRIA), Valerie Issarny (INRIA), Fabio Kon (USP), Pierre Chatel (THA), Amira Ben Hamida (EBM), Carlos Eduardo Moreira dos Santos (USP), Rafael Correia (USP), Daniel Cukier (USP), Nelson Lago (USP), Dionisis Athanasopoulos (UOI), Panos Vassiliadis (UOI).
Reviewer(s)	:	Marco Autili (UDA)

Abstract

This deliverable specifies the main concepts of the CHOReOS middleware architecture. Starting from the Future Internet (FI) challenges for scalability, heterogeneity, mobility, awareness, and adaptation that have been investigated in prior work done in WP1, we introduce the aforementioned concepts to deal with the requirements derived from the FI challenges.

In particular, we propose an extensible and scalable service discovery approach for the organization and discovery of services that relies on multiple service discovery protocols. Moreover, we introduce an extensible and scalable approach, based on the service bus paradigm, for service access that features the integration and adaptation of multiple interaction protocols. Furthermore, we propose solutions that enable the execution of FI service compositions that range from compositions of choreographed services, developed according to the CHOReOS development process, to massive compositions of things. Finally, we detail the Cloud & Grid middleware facilities that support the overall middleware and the choreographies that are built on it, via a unified API that provides access to multiple cloud infrastructures (e.g., Amazon EC2, HP Open Cirrus, private clouds).

Keyword list

Middleware, Service Discovery, Service Composition, Service Access, Choreographies, Cloud Computing, Grid Computing.

Document History

Version	Changes	Author(s)
1.0	Outline	Apostolos Zarras, Valerie Issarny, Nikolaos Georgantas, Fabio Kon, Hugues Vincent
1.1	First input from WP3 partners on ** sections	Apostolos Zarras, Dionysis Athanasopoulos, Thiago Teixeira, Sara Hachem, Valerie Issarny, Pierre Chatel, Carlos Eduardo Moreira dos Santos, Rafael Correia, Daniel Cukier, Fabio Kon
1.2	Drafts Section 2, 3.1	Apostolos Zarras, Dionysis Athanasopoulos, Thiago Teixeira, Sara Hachem, Valerie Issarny, Pierre Chatel, Carlos Eduardo Moreira dos Santos, Rafael Correia, Daniel Cukier, Fabio Kon
1.3	Revised section 1, Integrated input from INRIA, EBM	Apostolos Zarras, Dionysis Athanasopoulos, Thiago Teixeira, Sara Hachem, Valerie Issarny, Pierre Chatel, Carlos Eduardo Moreira dos Santos, Rafael Correia, Daniel Cukier, Fabio Kon, Amira Ben Hamida
1.4	Integrated input from INRIA, EBM, USP	Apostolos Zarras, Dionysis Athanasopoulos, Thiago Teixeira, Sara Hachem, Valerie Issarny, Pierre Chatel, Carlos Eduardo Moreira dos Santos, Rafael Correia, Daniel Cukier, Fabio Kon, Amira Ben Hamida
1.5	Editing & changes to UML notations (passing from package to component view) Added responses to challenges for service discovery section	Apostolos Zarras
1.6	Some comments from USP	Nelson Lago, Fabio Kon
1.7	New version of Section 5.4	Pierre Chatel

1.8	New version of Section 6	Carlos Eduardo Moreira dos Santos, Rafael Correia, Daniel Cukier, Fabio Kon
1.9	New version of Section 2, 3 integration of EBM input & some editing. Merged "response to requirements" subsections with "requirements" subsections	Amira Ben Hamida, Apostolos Zarras
1.10	New version of Section 5, input for Things	Amira Ben Hamida, Thiago Teixeira, Sara Hachem
1.11	Appendix B added for Things	Sara Hachem
1.12	Restructuring	All
1.14	Revised input for IoT	Thiago Teixeira, Sara Hachem
2.0	Input for XSB & editing existing material	Nikolaos Georgantas
2.1	Editing to address internal review comments	Apostolos Zarras
2.2	Further editing & updates to chapter 5	Apostolos Zarras, Hugues Vincent, Pierre Chatel
2.3	Editing based on UDA comments	Apostolos Zarras
2.4	Editing based on Marco/Valérie comments	Amira Ben Hamida
2.5	Overall revision	Fabio Kon
2.6	Minor changes in the architecture	Apostolos Zarras
2.7	Minor revisions	Apostolos Zarras
3.0	Minor revisions	Apostolos Zarras

Document Review

Review	Date	Ver.	Reviewers	Comments
Outline	13/05/2011	1.0	All authors	Outline agreed
Draft	10/10/2011	2.0	All authors	Internal WP3 review draft
QA	21/10/2011	3.0	Marco Autili (UDA), Hugues Vincent (THA), Valerie Issarny (INRIA), Fabio Kon (USP)	Editorial comments
PTC	21/10/2011	A	PTC	

Glossary, acronyms & abbreviations

Item	Description
AoSBM	Abstraction-oriented Service Base Management
BPEL	Business Process Execution Language
C&E	Composition & Estimation
CS	Client/Server
DPWS	Device Profile for Web Services
DSB	Distributed Service Bus
EA	Extension Activity
EAI	Enterprise Application Integration
ESB	Enterprise Service Bus
FI	Future Internet
GA	Generic Application
IDRE	Integrated Development and Runtime Environment
IoBS	Internet of Business Services
IoS	Internet of Services
IoT	Internet of Things
IoTS	Internet of Things-based Services
KB	Knowledge Base
LSB	Lightweight Service Bus
PS	Publish/Subscribe
SCA	Service Component Architecture
SOA	Service Oriented Architecture
SOM	Service Oriented Middleware
TD	Things Discovery protocol
TS	Tuple Space
WP	Work Package
WSDL	Web Service Description Language
XSA	eXtensible Service Access
XSB	eXtensible Service Bus
XSC	eXecutable Service Composition
XSD	eXtensible Service Discovery

Table of Contents

1. Introduction

Service-Oriented Computing (SOC) is now largely accepted as a well-founded reference paradigm for Internet computing [PTDL07]. Under SOC, networked devices and their hosted applications are abstracted as autonomous loosely coupled services within a network of interacting service *providers* and *consumers* (a.k.a. *clients*). Middleware plays a key role in the realization of the aforementioned paradigm as it provides mechanisms that enable the registration of services that become available over time, the discovery of services, the composition of services and, finally, the access to services.

Still, despite the remarkable progress of the SOC paradigm and supporting technologies in the last ten years, new challenges now arise by the foreseen evolution of the Internet towards the Future Internet (FI). As discussed in detail in [D1.2], in the FI, the typical issues of scale, heterogeneity, mobility, awareness, and adaptability shall be magnified, calling for new middleware solutions that are able to cope with these issues.

Practically, the FI vision challenges all the aspects of SOC middleware. In particular, service discovery should allow the discovery of different kinds of services that range from business services (BSs) to Thing-based services (TSs), while coping with the anticipated scaling up of the amount of service providers, service consumers, and services. Service discovery should be able to integrate multiple discovery protocols and adapt the protocols used with respect to the particular service discovery situation (e.g., discovery of business services published in the Web, discovery of services available in a local network). Similarly, service access to different kinds of services should be provided, while dealing with the increasing amount of these services and the consumers that access them. Access to the services should be provided based on different interaction protocols that may be suitable for the different kinds of services (e.g., client-server access to business services, publish-subscribe/tuple-space access to Thing-based services), while providing an integrated view of these protocols and the ability to adapt them. Finally, service composition should be scalable, with respect to the number of entities involved, and adaptable so as to deal with unforeseen situations (e.g., changes in the quality of composed services).

In this context and according to the CHOReOS work-plan, the main research outcome of WP3 in M12 is the specification of the CHOReOS middleware architecture, which is further detailed in this deliverable. This architecture shall serve as a basis for the first implementation of the CHOReOS middleware components that is scheduled for M18. Naturally, the design and development of the CHOReOS middleware is an iterative process. Therefore, we anticipate that the proposed middleware architecture shall be further refined after we get experience with the first implementation of the middleware. The refined architecture shall constitute the blueprint for the second version of the CHOReOS middleware implementation, which is scheduled for M24. At M24, the proposed middleware architecture shall be further enhanced with the specification of the CHOReOS reconfiguration facilities. By M36, the implementation of the CHOReOS middleware shall be integrated in the CHOReOS IDRE and assessed using the CHOReOS use cases.

Motivated by the main FI challenges and requirements that have been discussed in [D1.2], the CHOReOS middleware aims at supporting the execution of large-scale service choreographies, which is the foreseen service composition paradigm of the FI. To this end, *the CHOReOS middleware provides mechanisms for multi-protocol service discovery, multi-protocol service access, choreography execution and adaptation.* The provided mechanisms focus on the two main domains that are tackled in CHOReOS: the Internet of Business Services (IoBS) and the Internet of Things-based Services (IoTS). Hereafter, we use the term Internet of Services (IoS) to refer to the union of these domains, which covers the widest possible variety of services that is expected in the FI. Last, but very important regarding the FI challenges and requirements, the CHOReOS middleware architecture

incorporates the Cloud & Grid middleware, which supports computational- and storage-intensive tasks performed by service discovery, service access, and choreography execution and adaptation mechanisms. Moreover, the Cloud/Grid infrastructure may also be used by the choreographies that are built on top of the CHOReOS middleware.

Concerning the specification of the CHOReOS middleware, we undertake the typical approach [PA06] that starts from the specification of a *conceptual view* of the middleware architecture, which highlights the main concepts of the CHOReOS middleware mechanisms that enable service discovery, service access, large-scale choreography execution, and Cloud/Grid computing. The conceptual view of the CHOReOS middleware architecture will be refined in subsequent versions into corresponding *technical design views* that reflect the implementation of the CHOReOS middleware. In accordance with the above, this deliverable starts by setting the overall challenges and requirements posed by the FI, which in particular relate to its foreseen ultra-large scale, heterogeneity, mobility, awareness, and adaptability.

The deliverable is then structured in relation to the essential functionalities of the CHOReOS middleware, i.e., service discovery, access, composition, and Cloud/Grid computing. Precisely, in Chapter 2, we provide the necessary background on our definition of the FI vision and major challenges and requirements that come along with it, which have been discussed in detail in [D1.2]. In Chapter 2, we further discuss the overall CHOReOS middleware architecture. Then, in Chapter 3 we focus on service discovery, in Chapter 4 we concentrate on service access, and in Chapter 5 we discuss choreography execution middleware. In Chapter 6, we provide details regarding the Cloud & Grid middleware that supports the CHOReOS service discovery, access, and choreography execution mechanisms. Finally, the conclusions that summarize the main contributions of this deliverable are presented in Chapter 7.

This deliverable is further accompanied by four appendices and a companion deliverable. Appendices A and B provide more details on the middleware mechanisms that target the IoTS domain. Appendix C provides technical insight on the SCA standard[1], which is employed by some of the proposed middleware mechanisms to support the execution of service compositions. Appendix D, provides a detailed view of the API offered by the Cloud & Grid middleware. Finally, the companion deliverable [D3.1-comp] contains a survey of the DPWS standard for accessing TSs.

[1] http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

2. CHOReOS SOM FI Requirements and Architecture

The overall CHOReOS Service-Oriented Middleware (SOM) architecture is aligned with the FI high-level challenges that were extensively discussed in [D1.2]. In this chapter, we summarize the main points derived from this study. Then, we present the basic concepts of the CHOReOS middleware architecture, which are further detailed in Chapters 3, 4, 5, and 6.

2.1. FI High-Level Requirements & Challenges

The FI has become the main focus of several research and development initiatives all over the world, including initiatives in the EU[2], USA[3], China[4], Korea[5], and Japan[6].

However, despite the great interest in the FI, no common definition of it has been adopted yet. Still, considering that the FI will result from the evolution of today's Internet, the FI can be defined as the union and cooperation of the *Internet of Content*, *Internet of Services*, and *Internet of Things*, supported by an expanding network infrastructure foundation. Those core domains, which we find already in today's Internet in a preliminary stage, are not fully established yet and will emerge with the foreseen evolution of services, content, things, and networks, as summarized in Table 2-1.

FI Constituent	Definition
Internet of Content	Content is any type and volume of media. Content may be pre-recorded, cached or live, static or dynamic, monolithic or modular. Content may be combined, mixed or aggregated to generate new content and media. It may vary from a few bits (e.g., the temperature that a sensor measured) to interactive multimedia sessions and immersive complex and multi-dimensional virtual/real worlds representations [Daras09].
Internet of Services	An umbrella term to describe several interacting phenomena that will shape the future of how services are provided and operated on the Internet. The Internet of Services also comprises the various sets of Internet Applications including pervasive/immersive/ ambient, industrial/manufacturing, vehicular/logistics, financial/ ePayment/eBusiness, power network control/eEnergy, eHealth, and eGovernment applications [ETP09].
Internet of Things	A global network infrastructure, linking physical and virtual objects through the exploitation of data capture and communication capabilities. This infrastructure includes existing and evolving Internet and network developments. It will offer specific object-identification, sensor and connection capability as the basis for the development of independent cooperative services and applications. These will be characterized by a high degree of autonomous data capture, event transfer, network connectivity and interoperability [CASAGRAS09].

Table 2-1.The FI constituents.

[2] http://www.future-internet.eu
[3] http://www.nets-find.net
[4] http://www.cstnet.net.cn/english/cngi/cngi.htm
[5] http://fif.kr
[6] http://akari-project.nict.go.jp/eng/overview.htm

The FI is setting significant challenges over the computing and networking environments in general and the middleware in particular, as it magnifies the features of the already challenging Internet of today (see [D1.2], [IGHZ11]). Specifically, key challenges posed by the FI relate to and are amplified by the highly correlated nature of the requirements for scalable, heterogeneous, mobile, aware, and adaptive Internet, summarized in Table 2-2.

FI Requirements	FI Challenges
Scalability	The Internets of Content, Services, and Things face scalability issues that should be handled by the underlying middleware infrastructure that supports them [IGHZ11]. These challenges derive from the increasing number, size, and quality of their networked entities, which is further exacerbated by the empowerment of users who are now becoming *prosumers* [Pereira08,ETP09,SZ09]. For instance, simply considering the Internet of Things, the large amount of new information available through things needs to be comprehensively managed and aggregated to provide useful services [ETP09].
Heterogeneity	The FI will be heterogeneous in many dimensions that should be handled by the middleware [ETP09, TSFH09]. These dimensions relate to the diversity witnessed in the following aspects [IGHZ11]: • Domains of service-based entities that are incorporated in the FI (business services, Thing-based services). • Standards and technologies derived from the various service-oriented paradigms involved in the FI (WS* services, RESTfull services). • Service discovery protocols related to the aforementioned domains, standards, and technologies. • Service access protocols related to the aforementioned domains, standards, and technologies. • Service composition models, that concern the aforementioned domains, standards, and technologies.
Mobility	Unlike the current Internet, the middleware should seamlessly integrate mobility in the design of the FI [IGHZ11]. Indeed, an essential challenge for the FI lies in the explicit design of service access protocols, service discovery protocols, and service composition models for a mobile wireless world given that the majority of the connected entities are now mobile.
Awareness & Adaptability	Awareness and related adaptability are common requirements over the middleware for sustaining the FI, be it at the service, content, or Things level. Issues to be addressed include [IGHZ11]: being aware and adapting to the service access protocols, the service discovery protocols, and the service composition models involved in different FI environments. Moreover, being aware and adapting to changes in the services that become available and to the services non-functional properties is another key issue for the middleware.

Table 2-2. Summary of the FI requirements and related challenges.

2.2. Overview of the CHOReOS Middleware Architecture

Despite the FI requirements and the novel challenges derived from these requirements, which were discussed in Section 2.1, the CHOReOS middleware still remains a service-oriented middleware (SOM). Therefore, it consists of mechanisms that facilitate the access to services, the discovery of services, and the composition of services. Nevertheless, even at this level of abstraction, the CHOReOS middleware architecture is characterized by two main features:

- The CHOReOS middleware targets two different but interrelated domains of services: business services and Thing-based services. Based on this inherent characteristic, the high-level architecture of the CHOReOS middleware comprises corresponding domain-specific mechanisms that support the discovery of services, the access to services, and the execution of service compositions (see Figure 2-1 for the domain-specific mechanisms of the CHOReOS middleware architecture). The specificities of the functionalities offered by the domain-specific mechanisms are hidden by corresponding unified "eXtensible" middleware mechanisms that unify the access to the domain-specific middleware mechanisms.
- Computationally- and storage-intensive tasks of both the middleware and the choreographies are supported by the CHOReOS Cloud & Grid middleware.

Specifically, Figure 2-1 provides an overall view of the CHOReOS middleware architecture. As discussed in Chapter 1, this is a conceptual view that focuses in the main features of the middleware architecture and their relations, rather than a concrete technical design that is perfectly inline with the CHOReOS middleware implementation. Concerning the notation used in Figure 2-1, components correspond to middleware mechanisms, which may consist of further components. Nodes denote different service domains, such as the IoBS and the IoTS, or supporting infrastructures. Some special symbols and stereotypes are used to depict discovery protocols and registries involved in these protocols. Finally, associations denote semantic relationships between the concepts of the middleware architecture.

The main constituents of the CHOReOS middleware are the following:

- The **eXtensible Service Discovery (XSD)** service provides functionalities for the organization and the discovery of available business and Thing-based services. To deal with the FI requirements that relate to heterogeneity, mobility, awareness, and adaptability, the XSD service enables the use of multiple service discovery protocols that range from legacy (for the discovery of business services), to Thing-based. On the one hand, to deal with scalability issues related to the increasing amount of available business services, the XSD employs the abstraction-oriented organization and discovery mechanisms (i.e., the mechanisms that constitute the AoSBM component) developed as part of WP2 (see [D2.1] for further details); these mechanisms operate on top of the legacy service discovery protocols. Moreover, to deal with scalability issues raised by the increasing amount of available Thing-based services, the XSD comprises a specialized Things Discovery protocol (TD) that leverages the concept of semantic abstractions for the organization of Thing-based services and provides facilities for the probabilistic registration and discovery of these services.
- Access to business and Thing-based services is realized by the mechanisms of the **eXtensible Service Access (XSA)**. Specifically, to deal with the FI requirements that relate to heterogeneity and mobility, an abstract **eXtensible Service Bus (XSB)** provides means for the integration of multiple interaction protocols that range from protocols suitable for the interaction with business services to protocols suitable for the interaction with Things-based services. The integration of different protocols and the adaptation from one protocol to another, is based on a unified set of interaction primitives that constitute the Generic Application (GA) connector model presented in [D1.3]. The XSB is realized by corresponding concrete buses. Specifically, the

Distributed Service Bus (DSB) targets business services, while the **Light Service Bus (LSB)** targets Thing-based services. The unified interaction primitives serve as a bridge for deriving pair-wise mappings between the semantics of different interaction protocols. Technically, these mappings are realized by GA Bridge and Adapter elements.

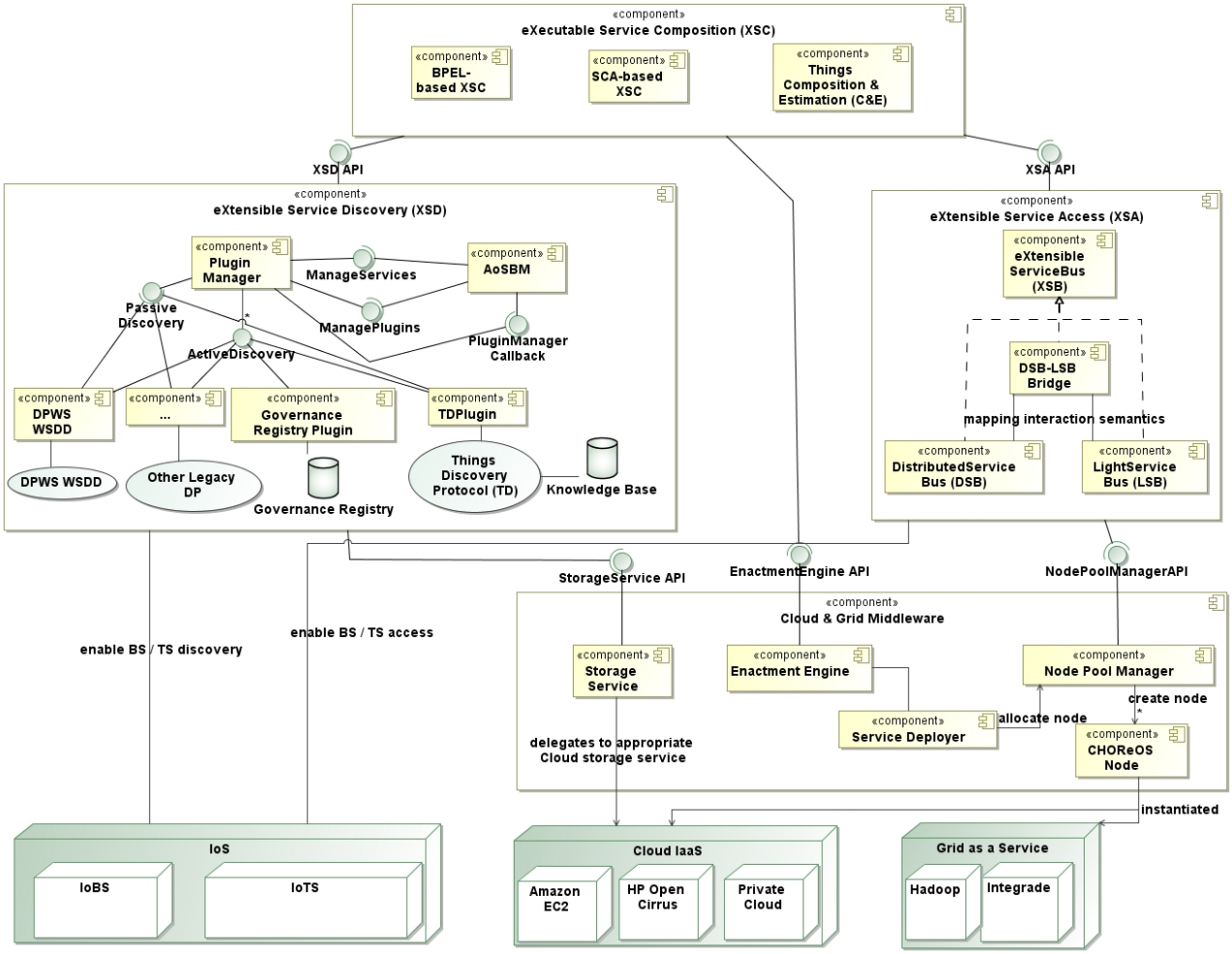

Figure 2-1. CHOReOS Middleware Architecture.

- The mechanisms that enable the execution of business and Thing-based service compositions are encapsulated in the **eXecutable Service Composition (XSC)** element. In particular, the XSC comprises mechanisms that realize the mapping of choreography models developed according to the CHOReOS development process defined in WP2 and described in [D2.1]. Moreover, XSC enables the execution of massive compositions of Thing-based services via the Composition & Estimation (C&E) component. The C&E focuses specifically on the scalability issues introduced by the need to compose an ultra-large number of Thing-based services towards calculating a required outcome. To deal with these issues, it relies on the concept of approximate composition.
- General scalability issues are handled by the **Cloud & Grid middleware**, which supports all the other middleware mechanisms that lie on top of it, by providing an execution platform, which can scale as the resource demands on the upper layers increase. However, the Cloud & Grid middleware is not yet another cloud infrastructure. Instead, it is a middleware layer that provides unified access to multiple cloud infrastructures (e.g., Amazon EC2, HP Open Cirrus, private clouds) under a common API.

Based on this overall conceptual view that introduces the main concepts of the CHOReOS middleware, the next subsections focus, respectively, on the main concepts of the mechanisms for the discovery, access, and composition of business services and on the main concepts of the mechanisms that target discovery, access, and composition of Thing-based services. Finally, we introduce the contribution that relates to the Cloud & Grid middleware.

2.2.1. The CHOReOS Middleware for Business Services

As mentioned previously, the key ideas for dealing with the FI requirements in the context of IoBS are: the use of abstractions and multiple protocols to enable business service discovery and access and the use of Cloud & Grid computing to support the execution of scalable choreographed business services.

More specifically, the contributions of the CHOReOS IoBS middleware are:

- **Business Service Discovery:** To facilitate the querying and the browsing of the constantly increasing amount of business services that are expected in the FI, the CHOReOS XSD employs the use of clustering mechanisms that allow to group services, which provide similar functional/non-functional properties and construct functional/non-functional abstractions that represent the aforementioned groups of services. As already discussed, these clustering mechanisms along with the proposed querying and browsing facilities are part of the AoSBM, developed in WP2 [D2.1]. However, a main issue that should be addressed to enable the abstractions-oriented organization of available business services is to provide a flexible mechanism that is capable of finding, and feeding to the AoSBM, information about the business services, which become available over time. Dealing with this issue amounts to putting in action the multi-protocol service discovery capabilities of the XSD. The discovery protocols are integrated in the XSD via a customizable plug-in architecture that can be configured and adapted with respect to the particular discovery situation based on a dedicated Plug-in Manager (Figure 2-1). The plug-in architecture may be used along with legacy plug-ins to retrieve information about available business services published in well-known service portals, in local networks, etc. Nevertheless, information about these services may be imprecise and unreliable. To deal with this issue the plug-in architecture comprises a plug-in that is dedicated to connect the XSD with the CHOReOS Governance registry, which provides information about business services that are controlled by the CHOReOS Governance framework developed in WP4 [D4.1].
- **Business Service Access:** To enable access to business services, the CHOReOS DSB relies on the PEtALS ESB (Enterprise Service Bus) technology, developed by the EBM partner. In particular, we aim at two complementary incarnations of the DSB. The first one, called **native-DSB**, employs the CHOReOS Cloud middleware to support the execution of the basic interaction protocols that are currently provided by the PEtALS ESB. The native-DSB avoids the overhead of supporting multi-protocol adaptation and consequently does not deal with choreographies where such support is needed. On the other hand, the extended DSB version, called **XSB-over-DSB** focuses on choreographies with requirements for interaction heterogeneity. To this end, the XSB-over-DSB adds to the PEtALS ESB the capabilities of adapting among multiple interaction protocols, based on the GA interaction primitives.
- **Business Service Composition:** Concerning the executable mapping of business choreography models developed according to the WP2 CHOReOS development process, we also consider two complementary approaches (Figure 2-1) that rely on corresponding integration technologies, which are popular in industry. The first approach, called **BPEL-based XSC**, builds upon the ESB technology (specifically the PEtALS ESB) and maps a choreography model and specifically the coordination delegates, derived according to the choreography synthesis method as part of the

CHOReOS development process, into distributed BPEL scripts. The second approach, called **SCA-based XSC**, relies on the SCA technology (specifically the FraSCAti SCA implementation) [SMFD09] and consequently maps the coordination delegates into SCA composite components.

2.2.2. The CHOReOS Middleware for Things

Many of the challenges related to the Internet of Things are directly inherited from the existing Internet, and therefore it is natural that some approaches that have been proposed for one can be adapted for the other. An approach that makes natural sense for the IoT is that of the Internet of Things-based Services (IoTS), which employs *service-orientation* to abstract each *thing* as a service. However, in some key areas, we find that we must make drastic changes as direct consequence of the IoT *physical* aspect, which does not exist in the current Internet. These changes are discussed in detail in the following.

First, the very *scale* of IoTS will necessarily be orders of magnitude greater than the existing Internet, since the IoTS requires *things* to be spread over large geographical areas, in spatially dense arrangements. This clearly differs from IoBS where business services, do not have a meaningful geographical aspect and therefore need not be duplicated all over the world (except for QoS purposes). Second, the IoTS also displays a trait that we call a *deep heterogeneity*: more so than business services, in the IoTS it makes a difference that *things* are produced by an assortment of vendors, with highly varying sensing/actuating characteristics, such as error distributions, sampling rates, spatial resolution, and so on. Many of these variations arise from differences in the manufacturing processes, in the exact hardware design, in the quality of materials, etc. — all of which are issues of the *physical world* and are absent in the realm of software-only *services*.

Finally, the IoTS also brings with it all the problems related to the *interpretation* of physical information. There are entire scientific fields devoted to making sense of the physical world (signal processing, estimation theory, robotics, etc.), and if *things* were simply handled as common *services* (without any special treatment) this would demand that application programmers assume the role of domain experts to use them to their fullest.

We address these issues in the CHOReOS IoTS middleware, which is discussed in Appendix A. This middleware takes sensing and actuation requests as input, and returns a desired result as output. At the highest level, its architecture consists of three parts incorporated in the overall CHOReOS middleware as shown in Figure 2-1: the **Things Discovery (TD)** protocol, the **Knowledge Base (KB)** and the **Composition & Estimation (C&E)** component. The element that is directly in charge of processing incoming requests is the C&E. In this process, the C&E must interact with both the KB and the TD to parse the incoming requests and generate the compositions of services that resolve it. Our contributions for each of these components are the following:

- **Things Discovery:** In a network where each discovery query may return millions of matching *things*, traditional discovery techniques become too onerous on the network infrastructure. To address this, we propose **probabilistic discovery** as a way to leverage well-known statistical properties of physical quantities and return only a much reduced — but still representative — subset of all matching *things.*

  This heavily relies on the **Knowledge Base,** which consists of three parts: a domain ontology (containing structured physical knowledge), a device ontology (carrying information about different sensors and actuators), and an estimation ontology (carrying mathematical methods for data estimation and interpretation). More details on the Knowledge Base are provided in Appendix B.

- **Things Access**: We propose six interaction paradigms that can be used to access sensors and actuators. More concretely, *thing* access can be (1) instantaneous, to perform a one-time sensing/actuating request; (2) periodic, to return a reply or execute

a command at a constant rate; or (3) event-based, as *things* may return a reply after being triggered by a specific event only (or controlled by sensors in the actuation case).

- **Things Composition and Estimation.** Similarly to the discovery case, the massive scale and heterogeneity of the IoTS make it unrealistic to look for an *optimal* solution within the combinatorially-large number of different service compositions that can answer a given query. As such, in our middleware, we instead pursue an **approximately-optimal composition**. The idea is to reduce the problem space considerably and still come up with a solution that is equally useful for the application that originally requested it. Furthermore, to address the previously-mentioned challenges of data interpretation, the C&E component is also in charge of **automatic estimation**. That is, by applying physical/statistical models on the historical dataset from surrounding *things*, the C&E component is able to estimate the most likely true value of the data at any given spatiotemporal point — whether there was a sensor at that exact location or not.

Meanwhile, in a proof-of-concept implementation, the CHOReOS IoTS middleware is implemented for Android- or Java-SE-compatible *things* or gateways and provides a unified interface for two aspects of *thing* interactions (registration and access) in addition to a things lookup interface. Furthermore, in terms of its actual hardware configuration, the CHOReOS IoTS middleware follows a dual approach where:

- Things that are programmable and have processing capabilities (such as mobile phones and smart sensor nodes) allow us to install software to directly interact with them.
- Other Things that use legacy/proprietary protocols to communicate in a closed network are interfaced by adapting their existing gateway device to interact with the IoTS middleware.

2.2.3. Cloud computing to sustain FI scale

Cloud computing plays a significant role for the CHOReOS middleware as it provides an execution platform that enables scalability for CHOReOS choreographies. In addition, research in CHOReOS will extend the state-of-the-art in Cloud computing by providing a novel mechanism for the execution of complex, distributed SOA systems on the Cloud.

The CHOReOS Cloud & Grid middleware will abstract different Infrastructure as a Service (IaaS) implementations in a common interface, providing a high-level API for the instantiation of choreographies. Thus, CHOReOS will act as a Platform as a Service (PaaS) system, enabling the enactment of choreographies that, if desirable, can work in the Software as a Service (SaaS) model.

Finally, CHOReOS will provide a novel solution for the execution of complex services on the Cloud. Current Cloud computing technologies focus on the allocation of virtual machines and on low-level resource management. The CHOReOS Cloud & Grid middleware, on the other hand, will raise the level of abstraction by providing the means to specify sophisticated distributed systems composed of multiple heterogeneous services in the end-user application level. The CHOReOS Cloud & Grid middleware will then map this specification to a real, large-scale distributed system that will be validated, automatically deployed, and executed on Cloud machines, monitored, verified, and dynamically adapted if necessary.

In summary, this model will enable the enactment of large-scale choreographies envisioned by the CHOReOS project and will advance the state-of-the-art of Cloud computing. A detailed description of the components and services to be provided by the CHOReOS Cloud & Grid layer is provided in Chapter 6.

3. Service Discovery

In this chapter, we concentrate on the CHOReOS middleware mechanisms that deal with service discovery (Figure 3-1). As already mentioned in Section 2.2, the CHOReOS service discovery has to deal with three main challenges:

1) The heterogeneity of service discovery protocols that are widely in use in both the IoBS and the IoTS domains.
2) The ultra large populations of services and service consumers in both domains.
3) The advanced and diverse requirements in terms of service registration and lookup capabilities rose by different components and purposes of the CHOReOS Integrated Development and Runtime Environment (IDRE) described in [D5.2].

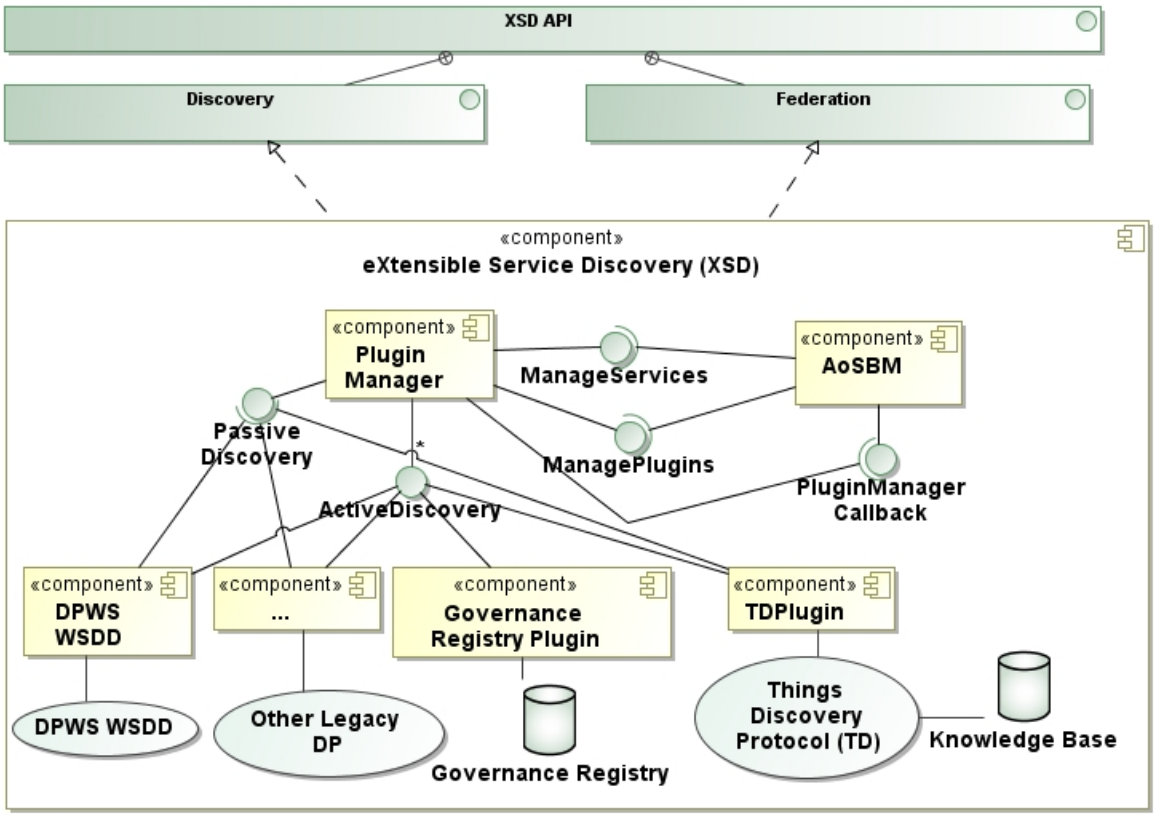

Figure 3-1. Service discovery mechanisms for the CHOReOS middleware.

Hence, we have opted for a highly extensible, scalable, customizable, and accessible solution for the CHOReOS service discovery, provided by the **eXtensible Service Discovery** service. XSD integrates the following features:

- XSD is exposed as a service, via the **XSD API** (comprising the **Discovery** and **Federation APIs** introduced below), so that it can be discovered, accessed, and composed as any other service. With respect to discovery, we envision an extensible distributed collaborative architecture for CHOReOS service discovery integrating multiple instances of XSD for ensuring scalability in the ultra large scale of FI. The definition of this architecture and the related Federation API will be done later in the course of the project.
- XSD is also internally extensible and customizable: it relies on a **plug-in** based architecture, which enables incorporating support for any current or future service discovery solution applying to IoBS and IoTS. XSD is adaptable, in the sense that the available discovery solutions can be adapted to different service discovery situations. As shown in Figure 3-1, plug-ins interact with service discovery protocols and/or service registries to collect information on services deployed in the related environments. To deal with the organization of services, the **Plug-in Manager**

coordinates all the different plug-ins and interacts with the **Abstraction-oriented Service Base Management (AoSBM)** to supply the **CHOReOS Abstraction Base** with the collected service information (see [D2.1] for further details). We detail our plug-in based solution in Section 3.1. In particular, we provide plug-ins for supporting certain legacy SDPs commonly found in IoBS and IoTS environments. Furthermore, we will develop a plug-in for incorporating the **CHOReOS Governance Registry**, which contains service information for all business services that are governed by the CHOReOS Governance framework and deployed on the DSB (see Section 4.2). We present this registry in Section 3.2. Finally, we will develop a plug-in for supporting the CHOReOS **Things Discovery** protocol, which is our solution to service discovery in IoTS. We introduce this protocol in Section 3.3.

- To accommodate the ultra large numbers of IoBS and IoTS services, XSD integrates a number of advanced techniques applying to the registration, storing, and lookup of service-related information. In particular, the **AoSBM** applies abstraction methods for organizing services in groups of similar services (see [D1.3, D2.1]), while the **Thing Discovery** protocol enables probabilistic registration and lookup (see Section 3.3).
- Finally, XSD provides a rich **Discovery** API, supporting advanced service lookup capabilities, which address the needs of the CHOReOS dynamic development process (WP2), the CHOReOS choreography execution middleware (WP3), and the CHOReOS governance and V&V (WP4). More specifically, this API supports the following features:
  - Governance-related lookup, where business services can be discovered based on properties of interest to governance (see Section 3.2).
  - Scalable lookup and browsing with respect to desired properties, which leverage the abstraction-based classification of services in the abstractions base (see [D1.3, D2.1]). Both functional and non-functional properties are supported.
  - Discrete and continuous lookup queries, where the latter enables receiving push-based discovery results over time whenever new discovery information is available. In Section 3.3, we explain more about the introduction of continuous lookup queries specifically in IoTS.
  - Probabilistic lookup queries for scalable discovery over large service populations. This is based on applying some statistical distribution function to a sought property to enable sampling a service population instead of browsing the entire population, nevertheless ensuring a reliable discovery result. In Section 3.3, we explain more about the introduction of probabilistic lookup queries specifically in IoTS.

3.1. Multi-Protocol Service Discovery

Our XSD plug-in architecture enables extensible multi-protocol service discovery. Our approach to plug-in based service discovery applies the following phases:

1. First, services advertise themselves by using a legacy protocol, such as UPnP SSDP[7] or DPWS WSDD [D3.1-Comp], or by registering in the Governance Registry, or by probabilistic registration employing the Thing Discovery protocol.
2. Next, service advertisements are discovered by the corresponding plug-ins and translated to a *common representation*. The representation is based on our definitions of component interfaces introduced in [D1.3] on the CHOReOS architectural style and on our definition of the Abstractions-oriented Service Base Management presented in [D2.1].

[7] http://www.upnp.org/

3. Descriptions of services, following the common representation are passed via the Plug-in Manager to the AoSBM. After being properly classified, information related to the classification of the service descriptions is stored in the Abstractions Base (see [D2.1] on the Abstractions Base).
4. Service lookup can then be performed based on the internal organization of service descriptions and matching mechanisms of the Abstractions Base and via the Discovery API (see [D2.1] on the Abstractions Base).

Our plug-in design builds upon existing platforms for SDPs and experience learned from them, in particular, the MUSDAC discovery platform [RICL07], the UbiSOAP discovery platform [CRI10], and the CONNECT Discovery Enabler [CONN-D1.2]. We extend these past-work solutions to deal with the advanced requirements of CHOReOS. More specifically, these solutions provide the runtime environment for plug-ins and a structured way for developing a new plug-in. Plug-ins have already been developed for the UPnP SSDP and DPWS WSDD protocols mentioned above. The set of plug-ins can be updated and extended dynamically by making changes to a configuration file, which is read by the Plug-in Manager.

The following two code snippets prescribe the API provided for developing plug-ins. The first one, is a part of the class definition of the Plug-in Manager (Table 3-1). Plug-ins interact with the Plug-in Manager via this class API. The second one (Table 3-2), is a part of the base class to be extended for developing a new plug-in.

Class PluginManager	
Methods	**Description**

`public static PluginManager getInstance(`
`  DiscoveryManagerInterface cbInterface`
`)`
`public static PluginManager getInstance()`
	Factory methods for getting a singleton instance of PluginManager.

`public void Terminate()`	Terminate the operation of the PluginManager.
`public void Restart()`	Restart the operation of the PluginManager.
`public boolean addPlugins()`	• Opens a file containing the list of plugin classes to load. Each line of the file follows a pattern like: `org.connect.enablers.discovery.plugins.cdp.CDPPlugin=ENABLED`. • Then iterates through the list calling `addPlugin (className)`. New plugins must be added to this file.

`public boolean addPlugin (`
`  String pluginClasseName`
`)`
	Construct plugin classes via reflection. Each plugin is stored in the hashmap (classname -> plugininstance). If the plugin implements an "active" rather than passive protocol, then the discovery request is initiated through calling `getNS()` on the plugin.

`public void delPlugin(`
`  String sdpName`
`)`
	Removes the plugin associated to the specified SDP.

`public void delPlugins()`	Deletes all plugins.

`public CNSState registerNS (`
`  DiscoveredNSDescription nsDescription,`
`  String nsLocation,`
`  Integer lifetime`
`)`
	• Called by plugins to place a discovered description (of a networked system NS) in the description repository.
	• Plugins get the PluginManager reference by calling the static method to get the instance

`public boolean updateNS (`
`  DiscoveredNSDescription nsDescription,`
`  Integer lifetime`
	Called by plugins to update a discovered description in the repository.

) public boolean unregisterNSbyUID(String serviceUuid)	Called by plugins to remove a discovered description from the repository.
public boolean unregisterNSbyAddress (String address)	Called by plugins to remove a discovered description from the repository.

Table 3-1. Plug-in Manager.

Class Plugin – plugins must extend this class	
Methods	**Description**
public Plugin(String pluginName, int supportedMode)	Constructor. The mode indicates active or passive. UPnP and DPWS are passive.
public Vector<DiscoveredNSDescription> getNS ()	Returns the services matching the provided service descriptions (if the plugin is active). NS stands for networked system.
public int getDiscoveryMode() public String getPluginName()	Returns information about the plugin
public boolean Terminate() public boolean Restart()	Terminate / Restart the operation of a plugin Plugins inform the manager of discovered NSs by calling PluginManager.registerNS () somewhere in their implementation.

Table 3-2. Plug-in.

3.2. Governance Registry for Business Services

The **Governance Registry** for business services enables the registration and discovery of services deployed on the DSB (Section 4.2) and fully supports their lifecycle from design to run-time. The need for covering the whole lifecycle comes from the fact that in ultra large-scale systems, information about services should be available at distinct phases of their lifecycle. Consequently, it is essential to provide a common and uniform abstraction level easing the service-related information retrieval and querying.

The Governance Registry enables governance activities such as V&V, SLA creation and negotiation and, finally, the runtime quality evaluation ([D4.1]). It provides functionality for testing, verifying, and validating a service before involving it in a choreography. Moreover, it makes it possible to look for services according to non-functional requirements and with regard to service level agreements.

Service consumers may access the governance registry functionalities via the XSD Discovery API and negotiate usage contracts. Once agreed, the service can participate in a choreography and be deployed on top of the XSB middleware, more precisely, on the DSB. At runtime, the governance framework serves for runtime quality evaluation to ensure that SLAs shall not be violated. This functionality is addressed by the CHOReOS Governance Framework presented in [D4.1].

In ULS systems, a significant number of services that are described in different and heterogeneous languages may be discovered. To tackle this issue, the Governance Registry for business services relies on a uniform service representation, which is inline with the model specified in [D1.3]. The latter is able to capture the most relevant information about services. Services are then published and discovered uniformly regardless of their initial respective descriptions.

Besides the governance-related capabilities, the Governance Registry enables the runtime discovery of the services deployed in the environment. To fulfil this, it is synchronized with the DSB.

The Governance Registry benefits from the PETaLS Naming Service that is internal to the DSB. As discussed in Section 4.2.1, the naming service maintains the relevant data about services deployed on the DSB. Service location, identification, and names are stored in a unique lightweight registry. This way, the bus identifies easily and efficiently service endpoints. The naming service is populated automatically, upon the deployment of new services on the DSB. It relies on the Java Business Integration specification and the APIs that are dedicated to handling the service discovery within the DSB middleware.

The naming service is populated as follows. When services are deployed on the DSB, a context is automatically created for them. Then, a reference endpoint is automatically created for each service. Once deployed, it is possible to activate and deactivate the service endpoint. Further, more advanced functionality is provided such as: endpoint registration, endpoints and data query and logging.

The CHOReOS project will provide an enhanced discovery protocol for business services based on the integration of both the naming service and the governance capabilities. Both will be integrated to provide an innovative synchronized view that conciliates the design time with the runtime environments. Indeed, we will be able to deploy services from the governance registry on top of a highly distributed service access middleware, and at the same time it will be possible to populate the registry automatically based on the data gathered from the several distributed runtime naming services of the DSB nodes.

Benefiting from the CHOReOS runtime middleware, the Governance Registry will enable the discovery of heterogeneous running services. Its discovery capabilities are augmented with V&V and quality runtime evaluation of services. Finally, further extensions shall be realized to enhance the naming service with awareness abilities. Actually, the naming service will be able to represent, in a common way, all services available in several DSB nodes. A peer-to-peer discovery approach could be adopted. Further studies will be realized to elucidate these contributions.

3.3. Thing Service Discovery Protocol

In our view, the existing state-of-the-art in service discovery is not sufficient to address the scalability and heterogeneity of the IoTS. While these approaches work well for the existing Internet (where traffic is made up of a relatively small amount of service interactions) they are not fit for the complex weave of interactions that will be commonplace in the IoTS. In the IoTS, a large number of requests will involve coordination among thousands of *things* and services, whereas on today's Internet most requests are largely point-to-point. Therefore, the number of packets transmitted in the network will grow strongly nonlinearly as the number of available services increases. In such an environment, performing even a simple service discovery may exceed acceptable time, processing, and memory constraints.

In our approach, we address these issues by introducing approximations and probabilistic properties into the discovery process, therefore leading to what we call **probabilistic discovery**.

Any discovery process can be defined as taking place in two phases: **registration** and **look-up**. *Registration* is the phase where each *thing* connects to a server (called a registry) to give some information about itself, including a network address and some relevant metadata. *Look-up* is when an external entity queries the discovery layer to find the *things*/services that match a given set of desirable metadata attributes (for instance, sensing modality, geographic location, error characteristics, etc.).

In probabilistic discovery realized by the TD protocol showed in Figure 2-1, we modify both

the registration and look-up phases to better support a massive, deeply heterogeneous network. These modifications can be summarized as follows:

- **Probabilistic Registration**: As billions of *things* are introduced into the network, the registries can quickly become overloaded with information. However, taking the physical aspect of *things* into account, one finds that much of this information is redundant, as neighboring devices can often replace one-another. Exploiting this fact, an approach often taken in the sensor-networking literature is to selectively utilize only a subset of all *things* at a time, in a process called duty-cycling. The idea behind probabilistic registration is inspired on the same principles, except that, in an unknown topology, there is likely no deterministic function that can oversee the duty-cycling process in an optimal manner. Therefore, we instead opt to pursue an approach where each *thing* uses a number of non-deterministic functions to effectively duty-cycle by registering/deregistering itself probabilistically at different times. The main research questions for probabilistic registration are which combinations of probability distributions should be used for each of the random variables above. For this purpose, the protocols used in traditional discovery methods do not necessarily have to change – instead, the only difference is *how* each *thing* uses these protocols. That is:
  - Which randomly-chosen registry or registries they should register with;
  - At what randomly-chosen times the registration process should take place;
  - What randomly-selected metadata attributes should be registered with each registry.
- **Probabilistic Look-Up:** We define as probabilistic look-up the act of querying the *things* repository to find the set of *things* and services that can *best approximate* the result that is being sought after. The main purpose of this process is to use predefined probability distributions to find approximate sets of services when exact solutions would be too costly to compute. For instance, if an application would like to find out the average temperature in Paris, the CHOReOS IoTS middleware should proceed by first fetching the definition of "average" from the Knowledge Base, which includes a description of the well-known equation for the sampling distribution of the mean. This equation states that in a network of M sensors, we can afford to instead use only N sensors ($N < M$) to calculate the average temperature within some mean error of e. With this information the discovery should, then, perform the following actions:
  1. Use the provided error equation to estimate the number N of sensors that will be needed for this request.
  2. Produce a random sample of N points in time, space, and other dimensions (such as sensor/actuator orientation in space, their coverage area, or any other attribute of a device).
  3. Discover N devices in the network that approximately match those N points.
  4. Given this set of devices, recalculate the error estimate.
  5. Repeat 2–5, depending on whether the new error estimate is satisfactory.
- **Continuous look-up:** In a traditional discovery protocol the look-up operation is instantaneous and *pull*-based. That is, an application usually sends a one-time query to a repository and pulls the query result containing all matching devices. However, for the IoTS, we propose an additional look-up approach, push-based, where an application continuously receives information about matching devices whenever the query results change over time. To illustrate this, consider an application that wishes to keep track of the temperature readings around a moving user. As the user changes his location, the relevant set of sensors will also change. Therefore, the CHOReOS IoTS middleware should continuously inform the application about any appropriate sensors discovered at the user's coordinates, as well as any sensors that have suddenly become too far away to be useful. Furthermore, the mobility of the application/user is not the only motivation

behind continuous look-up queries. Two other key reasons are: (1) the mobility of the *things* themselves, since the *things* in question could be moving cars, for instance; and (2) the dynamicity of the network, since new *things* that match a given look-up query may enter/leave the network at any point in time.

The probabilistic discovery approach should be implemented based on existing protocols. However, we are currently investigating which one of the existing protocols to adopt and implement as the Things Discovery protocol (among which is WSDD).

4. Service Access

In this chapter, we focus on the CHOReOS middleware mechanisms that address service access for both the IoBS and the IoTS domains in the FI (Figure 4-1).

As discussed in Section 2.2, the major issue for service access towards dealing with the FI requirements is to be able to cope with the diversity of interaction protocols involved in IoBS and IoTS and, specifically, the integration and the pair-wise adaptation of these protocols. To this end, the ESB paradigm is the established solution in Service Oriented Architecture (SOA) for dealing with interaction protocol diversity, due to its extensibility and scalability. Thus, we consider the ESB paradigm as the starting point in designing the CHOReOS mechanisms for service access.

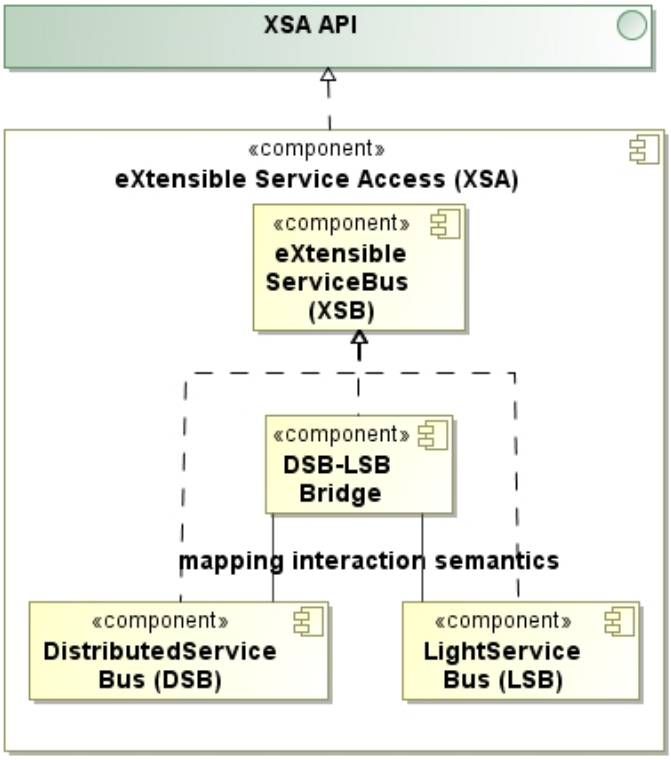

Figure 4-1. Service access mechanisms of the CHOReOS middleware.

In general, the ESB paradigm leverages best practices from EAI (Enterprise Application Integration) mechanisms and the service oriented architecture paradigm. ESB is based on an open, standard message backbone dedicated to enable the implementation, deployment and management of SOA-based systems. ESB exploits Web services, Message Oriented Middleware, smart message routing, and transformation mechanisms. In principle, ESB supports systems that involve a large number of services and high distribution and thus provides a scalable and manageable integration infrastructure. ESB acts as a mediator between service providers and consumers.

As depicted in Figure 4-2, ESB allows connecting applications, data sources, customer portals, and B2B interactions. Remote application integration is based on sophisticated mechanisms such as brokering, message transformation and routing, quality of service support, and service composition.

Services are discovered dynamically thanks to a common registry where service descriptions are stored and retrieved. The registry of the bus is technical in the sense that it stores the physical addresses of the services. The registry further holds meta-data related to providers and consumers. Service access related mechanisms allow heterogeneous applications and services to communicate with the bus. These mechanisms support heterogeneous protocols

and programming languages. Communication between services and applications is fulfilled through XML native messages. Messages are stored in a queue until their consumption by service consumers. Communication is further assisted by means of mediation patterns for routing, transformation, encoding, and mapping. The mediation patterns ensure the transformation of the message issued from an application to another application even if these are heterogeneous, as well as its encoding and its mapping to the right destination. Business processes and services are choreographed and orchestrated using a powerful engine. This engine is the cornerstone mechanism of integration in ESB solutions.

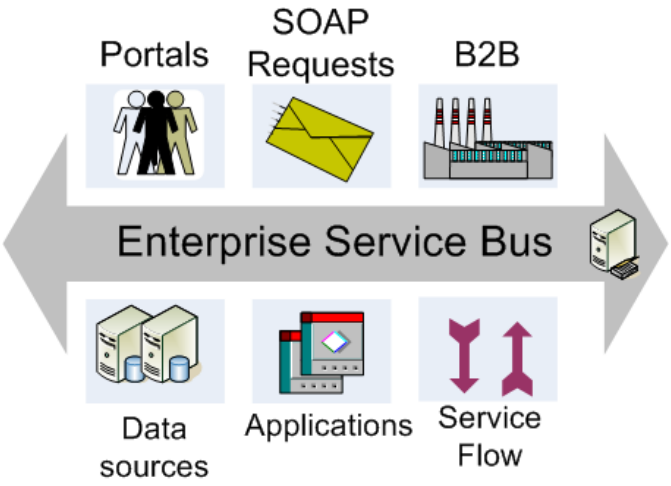

Figure 4-2. Functionalities of the Distributed Service Bus.

The main functionalities of an ESB are supported by components that can be both centralized or distributed over a network. All the above features make ESB a promising technological choice for Future Internet systems.

Nevertheless, CHOReOS aims to cover much larger interaction protocol diversity than the one commonly found in current SOA environments. In [D1.3], on the CHOReOS architectural style, we detail the CHOReOS vision for the connectors employed in the FI to interconnect Business Services and Thing-based Services. There, besides the typical *Client/Server (CS)* connector, we prescribe *Publish/Subscribe (PS)* and *Tuple Space (TS)* connectors, while we intend to add to these discrete connectors continuous ones, based on *streaming,* as well as, possibly additional connectors, representing other interaction paradigms; our objective is to be comprehensive. In this context, we claim that current ESBs fall short when it comes to supporting interoperability among such diverse interaction paradigms.

More specifically, common ESB implementations provide a common bus protocol that incorporates messaging semantics and is Web service-compatible, since SOA and Web services are the key targets of ESB-based infrastructures. This means that any system plugged into the bus is represented as a Web service, and its interaction protocol is mapped to message exchanges. Still, we have to note that the latest ESB implementations have included event-based communication to the bus protocol: this new semantics is supported in parallel with messaging [PH07]. In any case, we point out that common ESB semantics does not cover *cross-integration* of heterogeneous interaction paradigms, e.g., interoperability between CS and PS, or any other pair combination including TS, streaming, etc. The ESB solution in the example would be able to map both CS and PS to the *same* common bus semantics, i.e., choosing either messaging or events (when both are available) and map both CS and PS to the same single choice. This means that at least one mapping implies passing to a different semantics than the original one, which may introduce loss of semantics. Briefly stated, current ESB solutions are based on wrapping heterogeneous systems behind Web service interfaces, which, due to the loss of native semantics, hinders system integrators from accessing the fine-grained features of the individual systems being

integrated. The loss of native semantics may cause suboptimal functioning of the overall *system of systems*, i.e., the choreography of services in the CHOReOS context.

Hence, our solution to service access relies on an enhanced bus paradigm, the **eXtensible Service Bus** showed in Figure 4-1. XSB features richer interaction semantics than common ESB implementations to deal effectively with the increased IoBS/IoTS heterogeneity. Moreover, from its very conception, XSB incorporates special consideration for the cross-integration of heterogeneous interaction paradigms. When mapping between such paradigms, special attention is paid to the preservation of interaction semantics.

4.1. Multi-protocol Service Access

The XSB paradigm is based on our modeling of FI connectors presented in [D1.3] on the CHOReOS architectural style. More specifically, we introduced there a systematic abstraction of interaction paradigms, as depicted in Figure 4-3, with the following features:

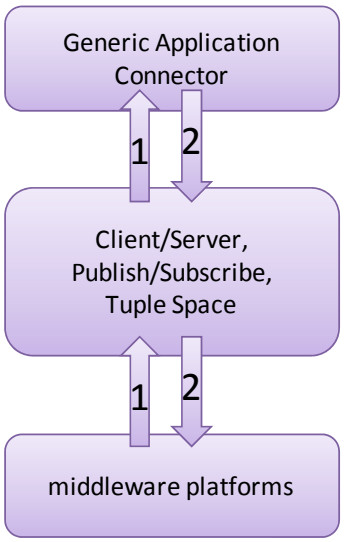

Figure 4-3. Abstraction of interaction paradigms.

- We abstract first from diverse middleware platforms (e.g., Web Services, JMS, LIME) to connector types representing the inherent interaction paradigms, i.e., CS, PS and TS (lower arrow numbered with 1) in Figure 4-3. As already mentioned, we intend to include streaming and others in these paradigms. In particular, we are currently surveying IoTS-oriented solutions, such as middleware for wireless sensor network architectures, to identify interaction paradigms in use in this domain (see Section 4.3). Our objective is to comprehensively cover both the IoBS and IoTS domains with our supported connector types.
- Then, we further abstract these connector types to a single higher-level connector type, which we call *Generic Application* (upper arrow numbered with 1). GA is a comprehensive connector type based on the union of the CS, PS, and TS connector types, where precise identification of the commonalities or similarities between the latter have enabled the optimization of the former. Further, GA preserves by construction the semantics of CS, PS, and TS. With the completion of CS, PS, and TS with more connector types, GA will represent a complete common abstraction solution for IoBS/IoTS.
- For both the above abstraction transformations, we provide counterpart concretizations, which enable transforming GA connector primitives to CS, PS, or TS connector primitives and then to concrete middleware platforms primitives (arrows numbered with 2).

- Furthermore, based on the GA abstraction we introduce mapping transformations between any pair from the set {CS, PS, TS} via GA. The fine knowledge of CS, PS, and TS semantics, as embedded in GA, enables these mappings to be precise: differing semantics are mapped to each other in such a way that loss of semantics is limited to the minimum. Thus, GA enables interoperability between heterogeneous interaction paradigms and, further, corresponding middleware platforms (sequence of arrows numbered with 1 and then 2).

Based on this abstraction approach, we introduce XSB as having the following features (see Figure 4-4):

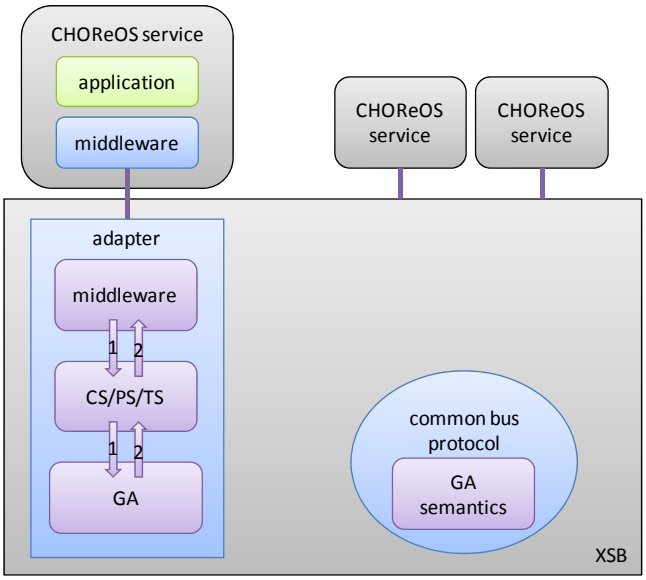

Figure 4-4. XSB support for multiple interaction protocols/paradigms.

- XSB can be seen as an abstract bus that prescribes only the high-level semantics of the common bus protocol. The XSB common bus protocol features GA semantics.
- Heterogeneous systems (or CHOReOS services) can be plugged into the XSB by employing adapters that adapt between the native middleware of the deployed system and the common bus protocol. This adaptation is based on the systematic abstraction discussed above (Figure 4-3), and in particular on the two-way transformation mappings between the native middleware platform, the corresponding CS/PS/TS abstraction, and the GA abstraction (arrows numbered with 1 and 2). Furthermore, adapters can resolve interoperability between heterogeneous interaction paradigms in the case where a GA primitive received from the bus represents an interaction paradigm other than the one of the native middleware platform.
- XSB, being an abstract bus, can have different implementations. This means that it needs to be complemented with a substrate which at least supports: (1) deployment (i.e., plugging) of various systems on the bus and (2) a common bus protocol implementing GA semantics. With respect to the latter, we envision that a GA protocol realization may either be designed and built from scratch (still supposing at least an IP-based transport substrate) or be implemented by conveying GA semantics on top of an existing higher-level protocol used as transport carrier. The latter solution can be attractive, as it facilitates GA protocol realizations in different contexts and domains.

Building upon the introduced XSB paradigm, the CHOReOS solution to service access lies in proposing a specialized bus proper to each one of the IoBS and IoTS domains, as an implementation of XSB:

- The **Distributed Service Bus** (Figure 4-1) targets the IoBS domain. DSB is based on PEtALS[8], which is a complete ESB solution brought by the EBM partner and is evolving in multiple ways within the CHOReOS project to support FI choreographies. To ensure flexibility and optimality with respect to different contexts and requirements, we have opted for providing two realizations of DSB:
    1. The **native-DSB** version, where the original PEtALS common bus protocol is preserved. This version applies to business service domains where interaction paradigms are rather homogeneous and mostly SOA-based. Thus, the extra processing overhead introduced by the GA semantics can be spared.
    2. An extended DSB version, which we call **XSB-over-DSB**, where the GA semantics is conveyed on top of the PEtALS common bus protocol. This version applies to business service domains that incorporate heterogeneous interaction paradigms. Thus, the enhanced GA semantics enables dealing effectively with the increased interaction heterogeneity.

  DSB is presented in Section 4.2.
- The **Lightweight Service Bus** (Figure 4-1) targets the IoTS domain. LSB is a lightweight bus introduced in CHOReOS, proper to IoTS specifics such as dynamics, resource constraints, data orientation, etc. From its very conception, LSB features GA semantics to deal with the increased IoTS interaction heterogeneity. LSB is presented in Section 4.3.

As stated from the beginning, CHOReOS service access primarily targets the integration of the IoBS and IoTS domains. Thus, we introduce further a bridging solution between DSB and LSB, which is naturally based – when applicable – on the common XSB heritage of the two buses. DSB-LSB bridging is presented in Section 4.4.

4.2. DSB for Business Services

DSB (Figure 4-1) is the CHOReOS service access solution for business services. In the following sections, we present the two identified realizations of DSB, native-DSB and XSB-over-DSB.

4.2.1. Native DSB

PEtALS is an Open Source ESB distributed by the OW2 middleware consortium under the LGPL license. It is built on top of the following agile technologies:

- The Java Business Integration (JBI) v1.0 specification, the Java standard for enterprise application integration. PEtALS DSB is fully compliant with the JBI specification.
- The Fractal Software Component Framework provided by the OW2 consortium. This framework is based on a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and graphical user interfaces. The PEtALS DSB implements the mechanisms that deal with the registration, message routing, message transport and discovery as Fractal components. This is a major feature, which allows core developers to specialize a PEtALS DSB distribution by choosing the software components they need.

[8] http://petals.ow2.org/

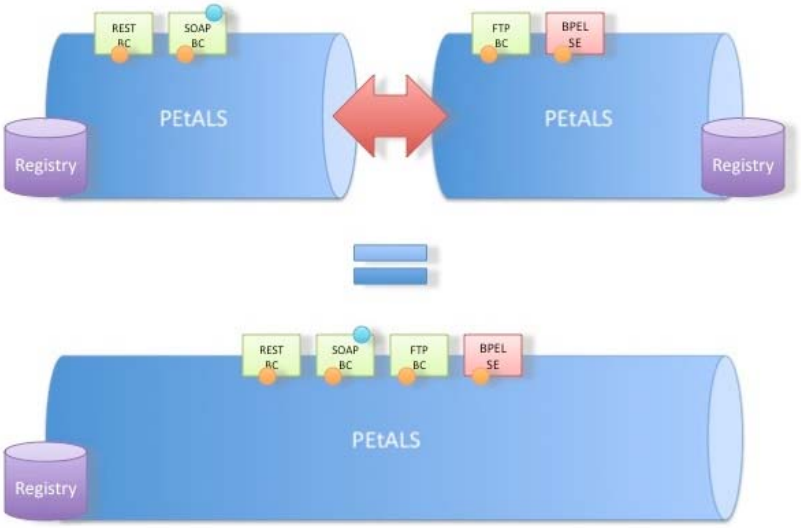

Figure 4-5. PEtALS Distributed Service Bus.

The PEtALS DSB (Figure 4-5) can be distributed across several processing nodes. These behave as a single unified PEtALS ESB node. Dealing with distribution in the PEtALS ESB was part of the work developed in the SOA4All FP7 EU project. The distribution is realized based on a transparent approach that ensures that all services remain accessible just as in a typical standalone service runtime environment.

Technical information related to service access, such as endpoints and container locations (physical network address) are managed by the PEtALS naming service. This service is used by the PEtALS container to deploy services in the PEtALS DSB and to route messages to the right endpoint. The naming service is replicated among all the PEtALS nodes using a Distributed Hash Table (DHT) over a multicast channel. This is equivalent to data flooding between networked entities, i.e., when an entry is added to the naming service, the data is sent to all the networked naming services. This way, all the naming services have a complete view of the services deployed in all the PEtALS containers. The routing of messages is realized by the PEtALS messaging mechanism, which implements the exchange of messages between distributed PEtALS containers. Indeed, In the PEtALS approach, once the endpoint is retrieved from the local naming service, the message and the endpoint reference are sent to the transport layer which is in charge of delivering the message to the endpoint regardless of its local or remote location.

In the context of CHOReOS, the PEtALS DSB shall evolve to the CHOReOS native-DSB so as to deal with the high-level FI requirements discussed in Section 2.1. More specifically, the evolution of the PEtALS DSB shall be towards the following innovations/developments:

- The CHOReOS native-DSB should be able to handle the constantly increasing amount of services and users that become available over time in the FI. To this end, the PEtALS DSB will exploit the CHOReOS Cloud middleware and benefit from its elasticity.
- The CHOReOS native-DSB must cope with highly distributed, mobile, and dynamic environments where services and users constantly appear/disappear. To deal with such environments, the PEtALS DSB will be enhanced with functionalities that allow it to be aware and adaptable to changes.

4.2.2. XSB over DSB

XSB-over-DSB is a complete DSB infrastructure, as presented in Section 4.1, where we enhance the common bus protocol with XSB semantics. In particular, GA semantics is

communicated on top of the PEtALS common bus protocol to enable interoperability among heterogeneous interaction paradigms, where preservation of semantics is a primary concern.

We have carried out an early realization of XSB-over-DSB. This work was done in collaboration with the ANR ITEmIS project[9], which shares some common interests with CHOReOS. In particular, we addressed the workflow-based orchestration of heterogeneous systems, which is a preliminary step before dealing with service choreographies. This work already provides a successful feasibility study of the XSB concept, as reported in the following.

The typical solution to system orchestration employs SOA workflow and ESB-based interoperability, as depicted in Figure 4-6 for a realization based on the PEtALS ESB. A number of systems are deployed or plugged on the bus via *binding components* – another term for adapters – which adapt the middleware platforms employed by the systems to the common bus protocol. Moreover, public native interface descriptions of the deployed systems are adapted to the common SOA/ESB-related interface description, which is most often a Web service description. Thus, plugged systems can identify each other and interact via the bus. Furthermore, ESBs typically support developing and deploying new applications that integrate plugged systems into orchestrations. This is much facilitated with the employment of a workflow language, most commonly BPEL. Application workflows provide application business logic and adapt between the potential application-level heterogeneity of the orchestrated systems. The application workflow can then be readily executed on a workflow engine embedded in the bus. Coordination between the workflow and the orchestrated systems is ensured by the interaction primitives of the workflow language, which are compatible with the common bus protocol: they are both Web-service compatible, i.e., they follow the CS interaction paradigm. As already pointed out, this results into undesirable loss of semantics when systems following heterogeneous interaction paradigms are integrated.

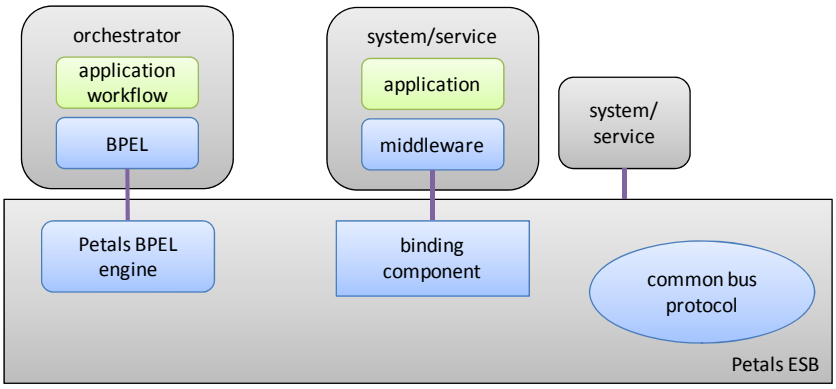

Figure 4-6. Workflow-based system orchestration on the PEtALS ESB.

We extend the typical SOA & ESB-based orchestration solution with support for multiple interaction paradigms, based on our abstraction approach, as depicted in Figure 4-7.

Second, at workflow execution time, encapsulated GA semantics are received and extracted by the binding component. Then, the binding component performs two successive transformations of the received GA primitives, first to the corresponding CS, PS or TS primitives, and then to the specific middleware platform primitives of the plugged system, by applying the transformation mappings discussed in Section 4.1. By employing the necessary

[9] www.itemis-anr.org

middleware library, the adapter can then interact with the system over its native middleware platform.

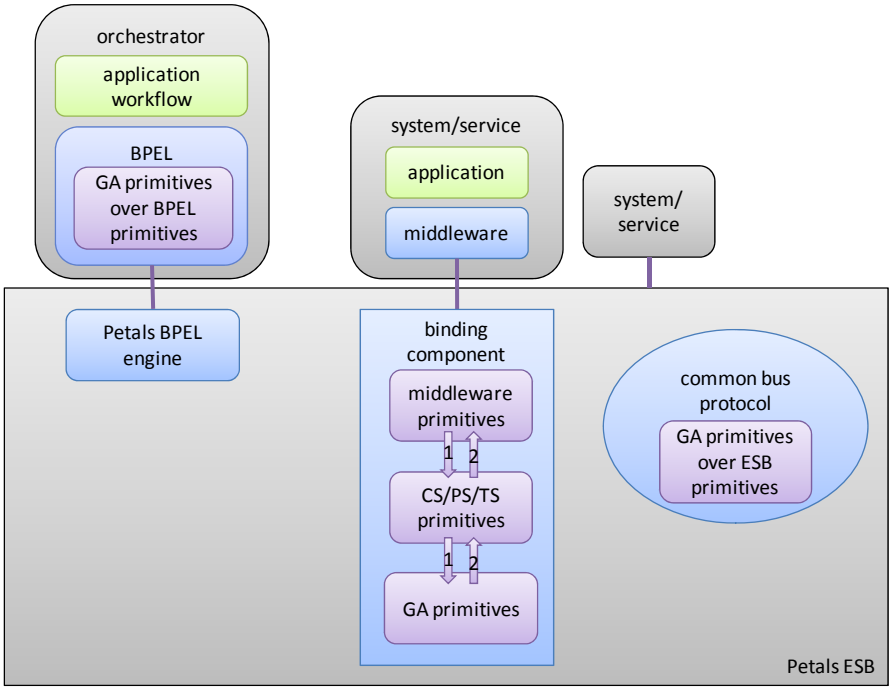

Figure 4-7. Extended orchestration supporting heterogeneous interaction paradigms.

First, we extend the BPEL workflow language with GA API primitives (see the definition of GA in [D1.3] on the CHOReOS architectural style). This enables the application developer to develop a workflow by employing the advanced GA API. We further introduce transformation between the GA-extended BPEL and the standard BPEL. This transformation consists in encapsulating GA primitives into standard BPEL primitives. This enables conveying GA semantics on top of BPEL primitives and subsequently on top of the common bus protocol primitives. Thus, we manage to communicate the advanced GA semantics without altering the standard SOA/ESB orchestration infrastructure. In this way, we are actually applying the XSB concept on PEtALS; hence, this is an early realization of XSB over DSB. In particular, checking the GA API primitives introduced in [D1.3] we can easily see that:

- Primitives with only input parameters, such as *post() (sends out data in the networking environment)* and *end_set_get() (closes a reception channel)*, can be conveyed on top of outgoing one-way asynchronous messages of the PEtALS messaging protocol;
- Primitives with both input and return parameters, such as *set_get() (sets up reception resources at the connector)* and *get_sync() (executes synchronous reception of data)*, can be conveyed on top of two-way synchronous invocations; and
- *notify() (called by the connector, enables asynchronous reception)* can be conveyed on top of an incoming one-way asynchronous message.
- Third, besides transformations of primitives at runtime, transformations of system interface descriptions should be performed at design-time. Our early work towards prescribing CS, PS, TS and GA-related interface descriptions is outlined in [D1.3]. Based on this work, public native interface descriptions of the orchestrated systems are transformed to CS/PS/TS interface descriptions, and then the latter are transformed to GA interface descriptions. GA interface descriptions are used by the application developer when designing the GA-enabled workflow. CS/PS/TS interface descriptions are used by the binding component to retrieve information about the corresponding orchestrated system at runtime.

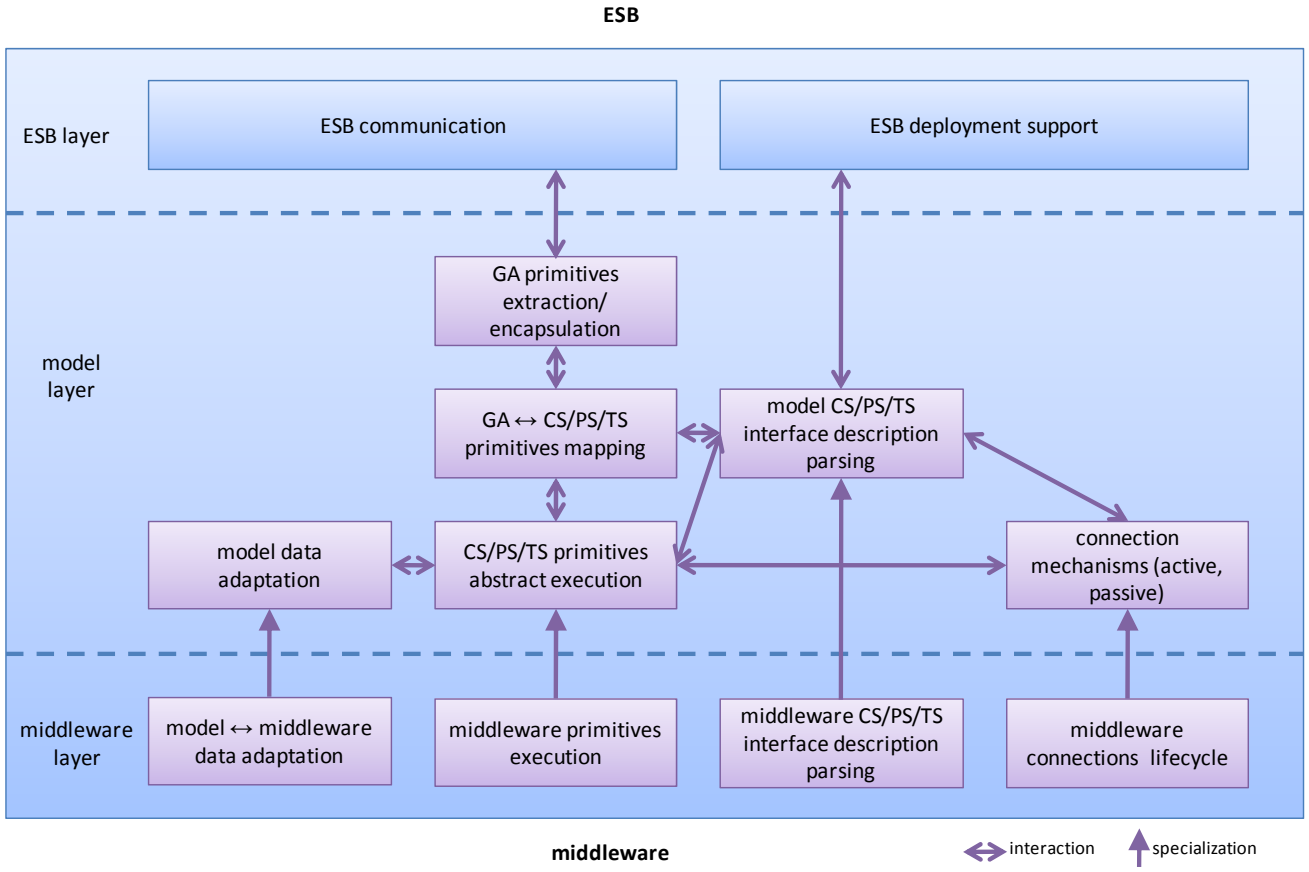

Figure 4-8. Architecture of the binding component.

The internal architecture of the binding component is depicted in more detail in Figure 4-8. On its upper part, the binding component communicates with the ESB, while on its lower part, it communicates with the middleware platform of the corresponding system. We distinguish three layers, as presented in the following. The *ESB layer* is devoted to the communication with the bus, and to the support of system deployment on the bus. These are standard functionalities, which are developed on top of the ESB APIs. The *model layer* concentrates the generic functionalities of the binding component, while the *middleware layer* provides functionality templates that need to be specialized for the specific middleware platform. Targeting facilitated extensibility of our solution, we provide a highly-optimized design, where the major part of the binding component functionalities are already implemented by the model layer, while a small part of additional specialization needs to be carried out at the middleware layer for introducing a new middleware platform.

The generic model-layer and specific middleware-layer functionalities form four columns, which we discuss hereafter. The second column from the left concerns processing of coordination primitives, and comprises, from top to bottom, extraction of GA primitives from the communication with the bus, mapping to CS/PS/TS primitives, abstract execution of these primitives, and finally execution of the actual middleware platform primitives. Only the latter template element needs to be completed for incorporating a new middleware platform. Upon primitives processing, the above functionality column interacts:

- With the first column, with regard to data adaptation between the specific middleware data types and the GA/CS/PS/TS model data types;
- With the third column, with regard to retrieving information from the CS/PS/TS interface description of the plugged system; and
- With the fourth column, with regard to managing the lifecycle of connections to the middleware platform of the plugged system.

Again, only the bottom-part, middleware-layer template elements need to be completed for a new middleware platform.

We provide next some more details about our implementation of the above architecture and of the overall support of extended orchestrations. We use the PEtALS inherent support for BPEL provided by the embedded EasyBPEL workflow engine. Furthermore, BPEL allows enhancing the base language with *extension activities (EAs)*. We exploit this powerful feature to implement the GA API primitives. More specifically, we introduce new BPEL EAs (in the form of XML Schema or XSD) representing the GA primitives. Regarding interface descriptions, we implement the CS/PS/TS/GA interface descriptions as XSDs, inspired from WSDL, the Web Services description language. Moreover, we propose two XSLT-based transformations [Kay04]:

- Transformation between GA-extended BPEL (i.e., the introduced EAs) and standard BPEL; and
- Transformation of CS/PS/TS interface descriptions to GA interface descriptions.

Finally, we implement in Java a generic binding component, which is the base for any binding component specialized for a middleware platform. For providing support for a new middleware platform, a developer additionally needs to:

- Specialize the base binding component for the specific middleware platform; and
- Introduce a transformation of native interface descriptions specific to this middleware platform to CS/PS/TS interface descriptions.

For designing an application workflow, the application developer needs to:

- Generate GA interface descriptions of the systems to be integrated; and
- Design a GA-enabled BPEL workflow, based on these descriptions, and then transform this workflow to standard executable BPEL workflow.

We have evaluated the applicability of our approach by implementing an orchestration workflow that integrates a Jini JavaSpaces TS system[10], a JMEDS DPWS Web Service[11], and a JMS PS system based on Apache ActiveMQ[12]. We have provided support for these three heterogeneous middleware platforms by specializing appropriate binding components, and have shown how our approach supports heterogeneous interaction paradigms on the PEtALS bus. Hence, we have demonstrated the feasibility of the XSB concept and provided an early realization of XSB-over-DSB. Besides showing feasibility, the further use of this early architecture and implementation can be two-fold:

1. The architecture and implementation of the binding component is perfectly valid for supporting heterogeneous service choreographies as well, besides service orchestrations. Nevertheless, we have supposed that an orchestration workflow interacts with a system deployed on the bus by employing the precise GA semantics that abstracts the interaction paradigm of this system. What needs to be additionally incorporated in the binding component architecture is the support for resolving interoperability between heterogeneous interaction paradigms (in the case where a GA primitive received on the bus by the binding component represents an interaction paradigm other than the one of the plugged native middleware platform, as discussed in Section 4.1). We prescribe the required mapping between heterogeneous interaction paradigms in Deliverable D1.3 [D1.3] on the CHOReOS architectural style.
2. Our solution to the incorporation of GA semantics in BPEL can serve as a guide for adding GA semantics to BPMN so as to enable designing service choreographies that

[10] http://www.jini.org/wiki/JavaSpaces_Specification
[11] http://ws4d.e-technik.uni-rostock.de/jmeds/
[12] http://activemq.apache.org/

can transparently accommodate heterogeneous interaction paradigms (see the CHOReOS vision on Connectors as well as Coordination Protocols in [D1.3]). Both GA-enhanced BPMN and BPEL can serve in designing and executing CHOReOS choreographies (see also Deliverable D2.1 [D2.1] on the CHOReOS dynamic development process).

4.3. LSB for Thing-Based Services

LSB (Figure 4-1) is the CHOReOS service access solution for Thing-based Services. We are introducing LSB from scratch in CHOReOS, and thus we are currently in the process of identifying:

- Interaction paradigms that are common in the IoTS domain and are not yet covered by the connector abstractions introduced in [D1.3] and discussed above in Section 4.1. Hence, we are currently surveying state-of-the-art protocols in IoTS-oriented solutions, such as middleware for wireless sensor network architectures. The first outcomes of this study confirm the importance and wide use of the already identified interaction paradigms, i.e., client/server [GTW10, PKGZ08], publish/subscribe [DGV09, TGDK10, HBD09, RG07, HTS08], and tuple space [CMMP07, WSBC04, FRL09], while database querying [MFHH05, NLZ07, GKKN03, CCDF03], combined or not with streaming [DLW08, NLZ07, GKKN03, HHLL03, CCDF03, KNLZ07] are additional paradigms that are largely used due to the data-oriented nature of IoTS-oriented solutions. Furthermore, Web protocols, as in the WS-* and REST approaches, seem to provide the right common integration infrastructure where all diverse interaction solutions can be plugged [GTW10, PKGZ08]. We still need to investigate the above more precisely. Our intention is to provide support for the new identified interaction paradigms in our connector abstractions, and hence provide support for them in the LSB. We note also that possible enhancements to our connector abstractions will apply globally to the XSB, and thus as well to the enhanced XSB-over-DSB.
- The CHOReOS access protocol for the IoTS, i.e., the protocol that we will develop for accessing mobile Thing-based Services based on the CHOReOS vision of the IoTS. We present our initial considerations for this protocol in Section 4.3.1. Certainly, we will provide support for this protocol in the LSB, thus making it interoperable with existing IoTS protocols.
- The LSB common bus protocol, which will enable interoperability among all IoTS-related access protocols. Relying on our initial survey of IoTS-oriented solutions outlined above, we present our initial considerations for this protocol in Section 4.3.2.

4.3.1. CHOReOS Protocol for Accessing Mobile Things

Of crucial importance to mobile things access is the existence of a consistent and well-defined set of interaction paradigms to support the interaction with *things*, so as to abstract away the low level details of the *things* specificities. As such, we consider that each *thing* necessarily implements at least one of the following:

- **Instantaneous sensing.** This is one of the simplest ways a *thing* can be accessed. It consists of simply querying a *thing* for its latest sensing value. For instance, consider an application that would like to find out whether or not it is windy at this moment. Then it may ask a nearby anemometer: “what is the wind speed right now?”. This returns a reply immediately.
- **Periodic sensing.** Examples of periodic sensing requests are: “record a 10s audio sample every minute”, and “measure the water level every hour”. This is an asynchronous request that returns a reply at a constant rate (until cancelled).
- **Event-based sensing.** Some sensors are inherently event-based (such as a motion sensor which sends an alert when movement is detected) while others are simply periodic sensors that have been coupled with a condition on its output (for instance, an

air pollution sensor that can be asked to transmit the detected pollution level if it exceeds some threshold value). Therefore, this is an asynchronous request that returns a reply at unpredictable times (until cancelled).

- **Instantaneous actuation.** Similarly to instantaneous sensing, instantaneous actuation consists of a one-time request to actuate a *thing*. For instance, a request to turn on the lights, to open a door, or to play a sound file.
- **Periodic actuation.** This is the actuation counterpart to periodic sensing. Some examples of periodic actuation are “starting the sprinklers on the lawn every night at 10PM”, “cooling down the freezer every hour”, “playing an alarm every morning at 8AM”.
- **Sensing-controlled actuation.** Many actuators can also report on their own status (or the status of the physical entity which they are actuating) by either performing the function of a sensor (many actuators are bidirectional transducers) or by including a discrete sensing component within the same hardware housing. For these, it is important to allow sensing-controlled actuation as a first-class operation to create simple control loops locally and avoid unnecessary network communications. Examples include “heating a room until the thermostat reaches 20C”, or “rotate the camera until the face-detector finds a face”.

As presented in Appendix A, we have implemented a preliminary prototype that is narrowed down to sensors reading within the same physical device (and especially smart-phone). This prototype will serve as a building block of the overall prototype of the CHOReOS Thing-based SOM.

4.3.2. Lightweight XSB

As introduced previously, LSB is a lightweight realization of XSB for the IoTS domain. The following list summarizes the features of LSB already pointed out:

- GA semantics.
- Fits to the IoTS specifics, in particular, dynamics and resource constraints.
- Enables interoperability among heterogeneous interaction paradigms abstracted by GA, including the paradigms and/or protocols discussed in Sections 4.3 and 4.3.1.

As prescribed in Section 4.1 for XSB realizations, the LSB core must support a deployment substrate for plugging Things and a common bus GA protocol; the latter is to be designed and built on top of an IP-based transport or a higher-level protocol transport. Targeting the IoTS domain and based on our ongoing survey of IoTS-oriented solutions that we briefly sketched in Section 4.3, it seems that a higher-level Web protocol and deployment infrastructure is most appropriate for the LSB core. We are considering DPWS as the most probable choice for the LSB common bus protocol. DPWS supports Web services on top of resource constrained devices and can provide a suitable transport for conveying GA semantics. We have carried out an extensive survey on the DPWS standard and its various existing implementations in a separate companion deliverable; the interested reader can check in [D3.1-Comp]. REST could be another possible choice. We are currently evaluating the advantages and disadvantages of both protocols to make our final choice. Another essential consideration is scale, which is a key concern in IoTS and the FI in general. We still need to evaluate this factor in our choice of the LSB common bus protocol.

4.4. DSB-LSB Bridging

DSB and LSB, as introduced before, realize service access in the IoBS and IoTS domains, respectively. DSB-LSB bridging (Figure 4-1) will enable the integration of the two domains in the FI as envisioned by CHOReOS. Based on the dual realization of DSB, i.e., as native-DSB and XSB-over-DSB, we consider three cases requiring a bridging solution, as depicted in Figure 4-9:

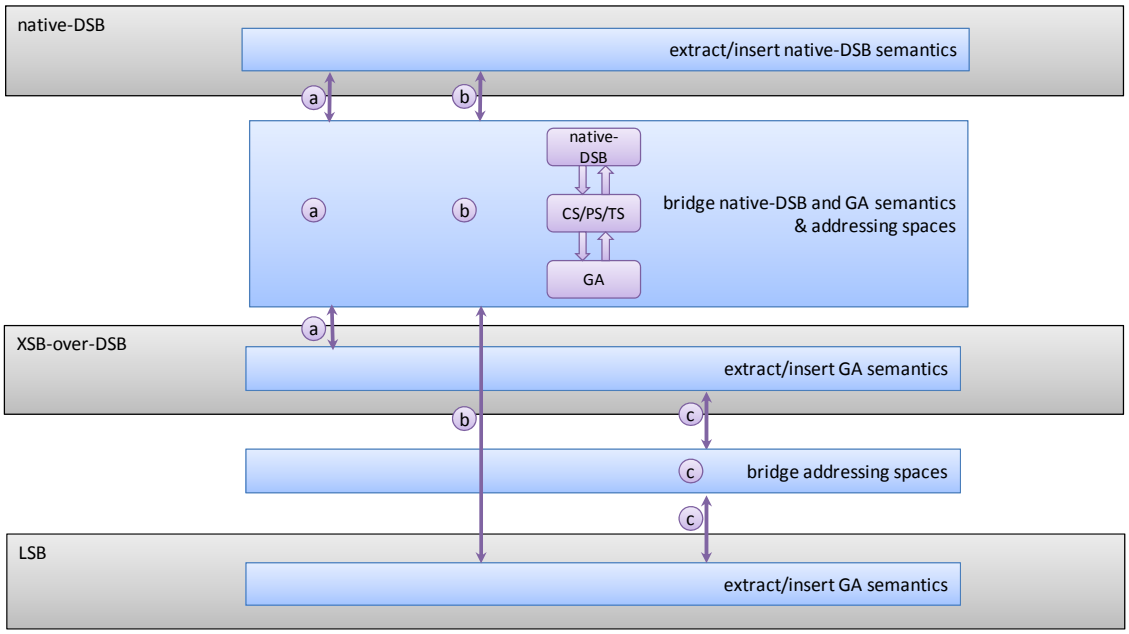

Figure 4-9. DSB-LSB Bridging.

a. **Native-DSB to XSB-over-DSB**. This bridging solution requires extracting/inserting native DSB semantics on the native-DSB bus, extracting/inserting GA semantics on the XSB-over-DSB bus, and bridging in the middle between native-DSB and GA semantics. The latter functionality is based on our abstraction of interaction paradigms discussed in Section 4.1 and depicted in Figure 4-3, as the native-DSB protocol can be considered as one more interaction paradigm covered by our abstractions. The bridging functionality also includes proper addressing space mapping between the two buses. Since both buses are DSBs, this is inherently tackled by the DSB federation feature, as presented in Section 4.2.1.

b. **Native-DSB to LSB**. The same bridging solution as in (a) applies here, where LSB takes the place of XSB-over-DSB. Nevertheless, addressing space mapping is not direct between the two buses, as they cover the two distinct IoBS and IoTS domains. We will devise an overall addressing space mapping solution for service access in CHOReOS when the two individual bus solutions, and in particular LSB, are settled.

c. **XSB-over-DSB to LSB**. This bridging solution is simpler, as both XSB-over-DSB and LSB directly feature GA semantics. Hence, it suffices to enable extracting/inserting GA semantics on both buses and then link them almost directly. Still, addressing space mapping as in (b) should be taken care of also here by a lightweight bridge in the middle.

The implementation and deployment of the above bridging solutions can take different forms and will rely on the deployment (plugging-in) facilities provided by DSB and LSB including adapters that adapt between a deployed component and the common bus protocol.

5. Executable Service Composition

In this chapter, we concentrate in the CHOReOS middleware mechanisms that enable the execution of service compositions. Specifically, XSC (shown in Figure 5-1) is the CHOReOS solution to the aforementioned goal. As already introduced in Section 2.2.1 the XSC comprises mechanisms that facilitate the execution of business choreographies developed according to the CHOReOS development process, specified in [D2.1]; these mechanisms are further detailed in Section 5.1. Moreover, as introduced in Section 2.2.2 the XSC comprises mechanisms that facilitate the execution of massive Thing-based service compositions; this part of XSC is detailed in Section 5.2.

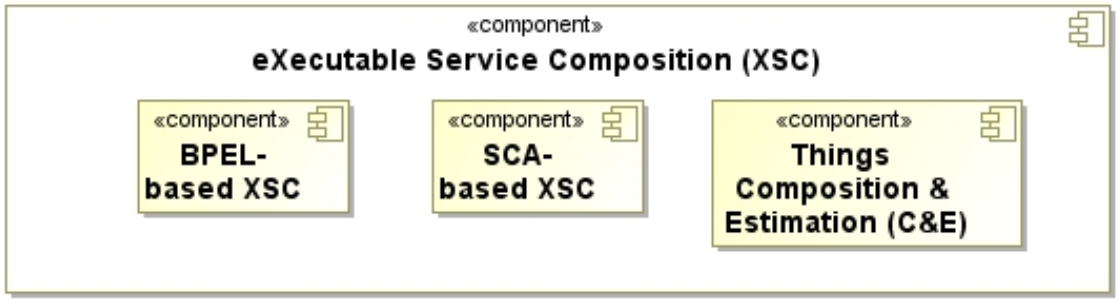

Figure 5-1. Service composition execution mechanisms of the CHOReOS middleware.

5.1. XSC for Business Choreographies

The main issue towards enabling the execution of business choreographies is to map them into a certain executable form. Specifically, the key point is to map the coordination delegates, derived from the CHOReOS development process [D2.1], to a certain executable form.

The coordination delegates play a main role in a choreography model as they encapsulate the coordination logic that is needed to coordinate the choreography participants, specifically the concrete services that are actually chosen (via the AoBM component of the CHOReOS XSD – Chapter 3) to play the roles of the participants (Figure 5-2). As reported in [D1.3] (see the beginning of Chapter 4), the problem solved by coordination delegates is to coordinate the global interaction behaviour of the participant services in order to guide their collaboration so as to fulfil the specified choreography. This calls for a suitable notion of coordination protocol since it might be the case that the collaborating services, although potentially suitable in isolation, when interacting together can lead to undesired interactions. The undesired interactions are those interactions, that can happen by letting the discovered participant services collaborate (in an uncontrolled way), which do not belong to the set of interactions modelled by the choreography specification. In fact, by introducing in the choreography-based system, the synthesized coordination delegates, hence controlling/supervising the collaboration of the participant services, we want to restrict the set of all possible interactions among the collaborating participant services to the set of those interactions modelled by the choreography specification.

As discussed in Section 2.2.2, in CHOReOS we consider two alternative mappings for coordination delegates, based on BPEL and SCA. These alternative mappings are realized, respectively, by the BPEL-based XSC which is discussed in Section 5.1.1 and the SCA-based XSC, which is discussed in Section 5.1.2. The main reason behind this choice is the increased interest for the aforementioned technologies in industry. However, both of the proposed solutions are inline with the same view concerning the mapping of choreographies models to a certain executable form. Based on this view, the two alternative XSCs take in charge of automatically, or semi-automatically translating a given choreography model, into an executable form, deploying the necessary elements of the executable choreography and further managing non-functional requirements that relate to the executable choreography.

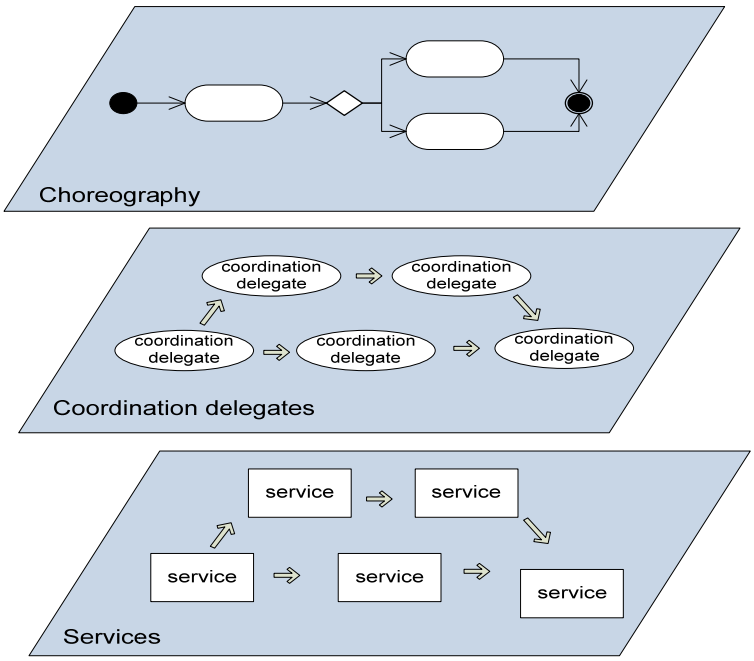

Figure 5-2. Mapping of choreography models.

5.1.1. BPEL-based XSC

BPEL-based Mapping of Coordination Delegates

Figure 5-3, highlights the main concepts of the BPEL-based mapping of business choreographies.

Starting from the choreography development process that synthesizes from the choreography specification a set of coordination delegates, the BPEL-based XSC interprets the coordination logic dictated by the coordination delegates and executes it. More specifically:

- Once the participants are identified and the coordination delegates implemented, the coordination delegates are deployed on top of the XSB middleware and precisely on top of the DSB.
- In an ultra large-scale context, the coordination delegates will be widely spread and distributed. Hence, to access each other they rely on the capabilities of the distributed service bus that allows reaching remote coordination delegates in multiple distributed service bus nodes.

As discussed in [D2.1], in CHOReOS, we use the BPMN2.0 language for the specification of choreographies. This language is able to express, in a synthetic way, the collaboration of remote partners involving several services. However, the coordination logic of coordination delegates should be defined in an executable language. The CHOReOS DSB embeds a BPEL engine that supports the execution of the coordination delegates since the language used for their specification can be mapped onto BPEL. Noteworthy, BPEL is a widely adopted and standardized language for developing coordination logic between business services. Regarding the service description, WSDL is also a widely used standard. Hence, the coordination delegates generated from the work in WP2 should be compliant with both of the aforementioned standards to support choreographies of business services. The coordination delegates may be extended to support other composition languages.

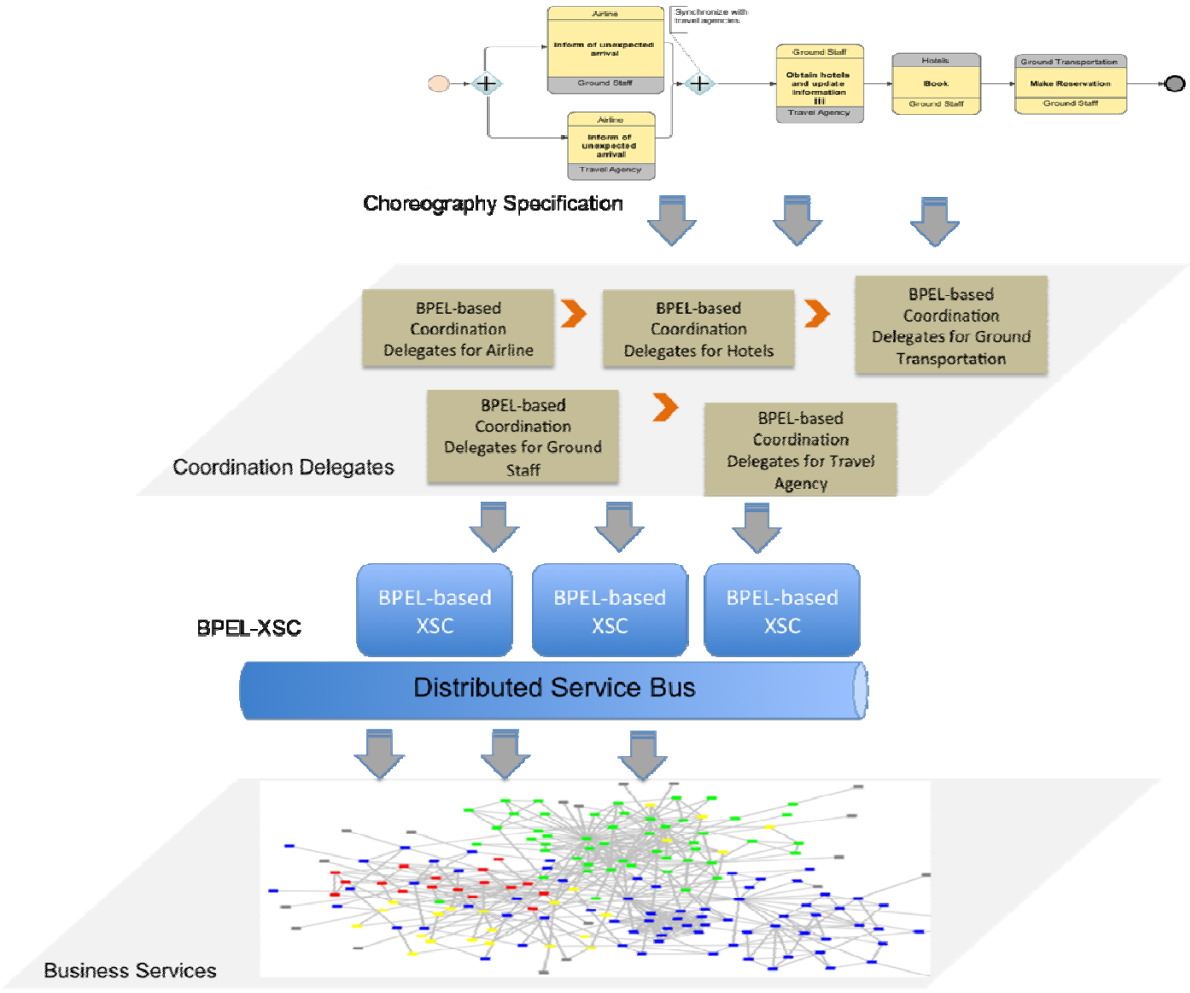

Figure 5-3. BPEL-based mapping of choreographies.

To support the coordination delegates, the CHOReOS DSB relies on dedicated libraries for both WSDL and BPEL for managing service descriptions and workflows, respectively. Each coordination delegate is interpreted as an executable service dependency graph, while the whole choreography is interpreted as a set of connected and distributed graphs of services. The XSC needs to cope with the new complexity brought by the choreography level.

Deployment & Execution

After being translated to BPEL, the coordination delegates are deployed on top of the DSB. Each BPEL-based implementation of the coordination delegate is interpreted as an executable graph. Indeed, collaborations within a choreography of services are considered as dependency graphs where the services are nodes and the message flows between them are arrows. From a conceptual point of view, the whole choreography is described as a complex graph. The deployment of this graph of graphs is done over a distributed infrastructure consisting of DSB enabled nodes. Each DSB node will host one or several coordination delegates. Benefiting from the DSB service access capabilities, the coordination delegates are able to realize the choreography in a highly distributed manner.

Business services involved in a given choreography model are also deployed and accessed on the DSB nodes. Messaging, routing, and transformation capabilities are ensured by the DSB capabilities as discussed in Section 4.2.

Management of Non-Functional Choreography Requirements

The management of non-functional choreography requirements is handled by the BPEL-based XSC in close collaboration with the Governance framework developed in WP4. In

particular, the business service monitoring capabilities of the Governance framework are employed to monitor the good behaviour of coordination delegates. This task involves a dedicated communication monitoring component that monitors the distributed delegates that execute a given choreography. Business service monitoring is responsible for detecting the violation of the choreography specification.

The provided monitoring functionality relies on an event-based architecture. Indeed, considering the BPEL-based mapping of coordination delegates, each coordination delegate within a choreography can be an event producer. The benefit from adopting an event-based architecture resides in its adaptation to a highly dynamic and distributed environment. Within CHOReOS, the communication monitoring service is in charge of monitoring the choreographies and, to this purpose, it subscribes to the events produced by each delegate. The events are agreed patterns described in the coordination delegates and expressing the collaboration as sending, receiving, loops, conditional structures, etc. This way the communication monitoring is notified by any violation of the expected behavior.

We briefly highlight the monitoring functionality in Figure 5-4. In this figure, we illustrate, in a layered view, the correspondence between the choreography model and the monitoring activities.

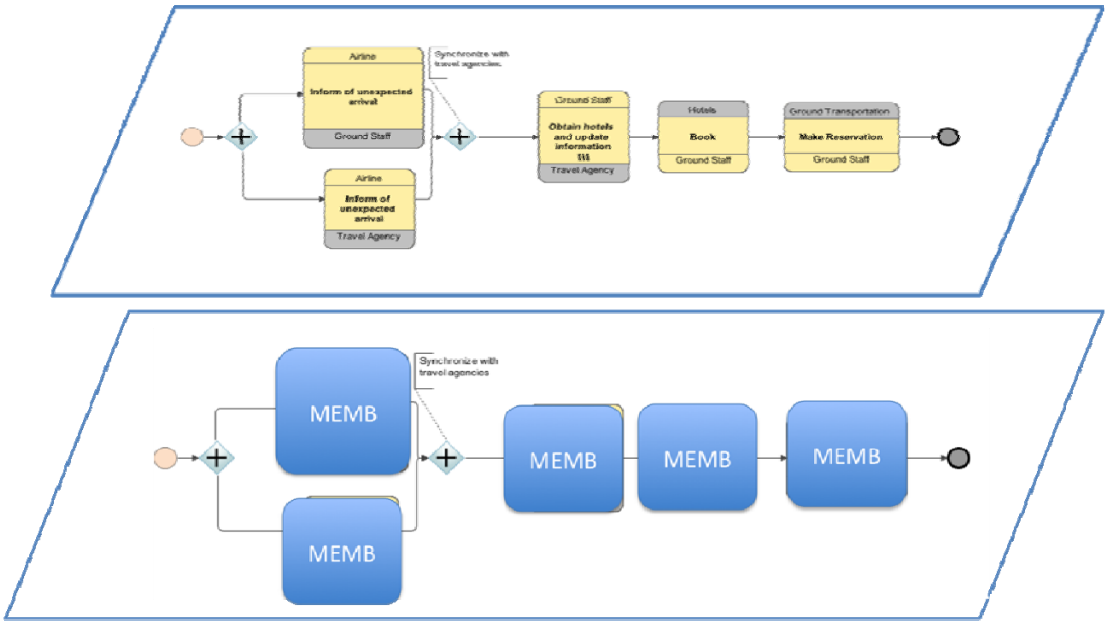

Figure 5-4. Monitoring the choreography services.

A specific structure called MEMB (for Message Exchange Monitoring Behavior) is dedicated to implementing the choreography monitoring; to this purpose, we assign to each coordination delegate a MEMB. Each MEMB subscribes to specific topics related to the coordination logic. Then, it waits for the expected notifications and checks the several timestamps validity. In case of timeout or not acceptable timestamps, alerts are sent to the Governance and V&V Framework.

5.1.2. SCA-based XSC

SCA-based Mapping of Coordination Delegates

Motivations to implement this SCA-based runtime range from environment-related constraints to more technical aspects. Indeed, from an industrial point-of-view (Thales being a leading industrial partner), SCA is of great importance since it allows economies of scale by fostering reuse in software products. As such, Thales leverages, in CHOReOS, its specific knowledge of SCA to implement the SCA-based XSC. Moreover, as illustrated in the context of the “Passenger-friendly Airport” scenario (see WP6 – [D6.1]), multiple legacy

systems may need to be integrated into CHOReOS choreographies. Relying on SCA eases the direct integration of these legacy systems (through SCA *bindings*) in the runtime. From a technical standpoint, compared to other component approaches such as OSGi, SCA has the advantage of being directly targeted at SOA-environments and designed to support a distributed communication and computation context.

Based on the SCA modular *binding* mechanism, the SCA-based XSC will be well integrated with other CHOReOS middleware mechanisms for the execution of service compositions and for service access. Indeed, Coordination delegates implemented over SCA will be able to target indistinctively and transparently business services and Thing-based services available directly or through the CHOReOS eXtensible Service Bus (XSB).

As highlighted in Figure 5-5, the SCA framework is directly leveraged to provide the realization of coordination delegates, as well as implementation of their encapsulated coordination logic. Moreover, the SCA framework is directly used to realize bindings between the coordination delegates and the targeted services: as shown in the figure, these services are then accessed through the XSB (composed of the DSB for business services, and LSB for Thing-based services) or directly, as made possible by any SCA container.

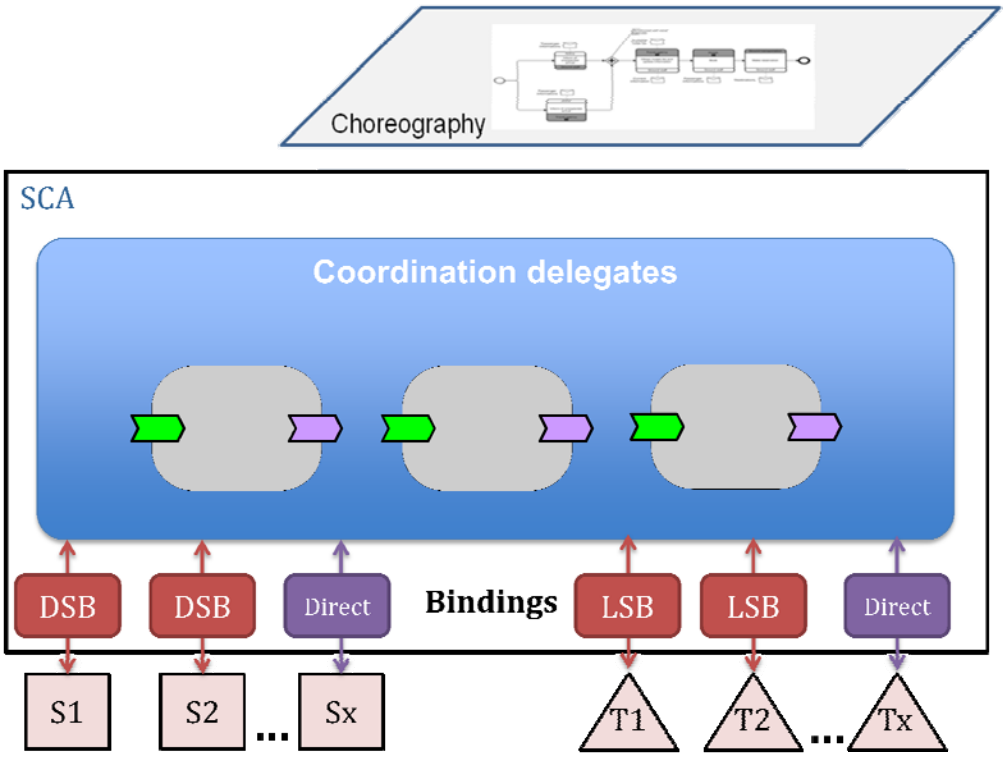

Figure 5-5. Use of Coordination delegates.

A more detailed view of the building blocks of the proposed SCA-based mapping is given in Figure 5-6. Specifically, according to the proposed mapping, the **coordination delegates** are implemented as **SCA composites**. Two distinct Coordination delegates are shown in Figure 5-6, both are connected to business (squares) and Thing-based (triangles) services. A **Coordination delegate** consists of:

- **The coordination logic** encapsulated that is implemented as BPEL-based SCA components[13] inside the composite; there is one BPEL-based component of this type per composite.
- **Simple and complex proxies** that realize the binding to the available services. These proxies, which are CHOReOS specific, relying on SCA bindings to link to actual

[13] BPEL-based components are a standard SCA feature. FraSCAti includes a BPEL runtime, as such a *separate dedicated BPEL runtime* is not necessary.

business and Thing-based services, **available through the XSB or directly** (as shown in Figure 5-6), at runtime. Binding between an SCA-based Coordination Delegate and other Coordination Delegates available on the XSB is made through the same DSB and/or LSB bindings. A key principle of the proposed mapping is that services are addressed transparently. As such, a coordination delegate can consume business services, Thing-based services, or both at the same time, depending on its requirements. This is made possible by the proxies. The respective role of simple and complex proxies is discussed below.

Both simple and complex proxies are implemented as standard SCA elements (components and composites) and have a dedicated purpose:

- **Simple proxies** are used to separate service access and protocol heterogeneity aspects from business logic aspects. To do so, as a middleman between a BPEL-based component and an actual service, a simple proxy exposes a SCA service whose interface matches a specific reference required from a business logic component. The business logic component is linked to this reference through an SCA wire. In other words, the functional requirements of the business logic component, expressed via the required reference, are mapped to a concrete service. The required reference is bound to a service at runtime, using the right SCA binding depending on the access mechanism and/or protocol used by the service.
- **Complex proxies** deal with protocol discrepancies when simple proxies are not enough. For instance when a functional requirement of a BPEL component cannot be *directly* satisfied by service available at runtime, the goal of a complex proxy is to compose multiple available services to match the functional requirement and expose a corresponding SCA service. The coordination delegate in the center of Figure 5-6 relies on a complex proxy to bind and combine S1 and S2 Web-services and provide a missing feature. The bottom part of the figure shows the internals of a complex proxy. It is implemented as a composite that uses simple proxies for binding to services and a Proxy logic component that composes features offered by those targets.

In a typical SCA fashion, the interfaces and references of proxies (which are realized as SCA composites) are **indirectly** connected to external systems (such as business services and Thing-based services), as can be seen in Figure 5-6. In particular, the SCA composites are linked to external systems, instead of the constituent elements of SCA composites. This relies on the *promotion* mechanism that exposes internal services and references. In our model, a generated composite (either Coordination delegate or Complex proxy) will expose its business logic / BPEL process through a single SCA service. Moreover, the generated composite may require multiple references, corresponding to the external targets that the generated composite needs to bind to.

Finally, it is to be noted that complex proxies bear a striking resemblance to coordination delegates, except for their two distinct internal components: business logic and proxy logic, respectively. This can be seen in Figure 5-6. At implementation time in WP3, we will thus study the possibility of directly implementing Complex proxies as Coordination delegates. If done right, our SCA mapping will then be greatly simplified and be more efficient. What would remain, though, is the conceptual distinction between both these constructs, used in our mapping for two distinct purposes.

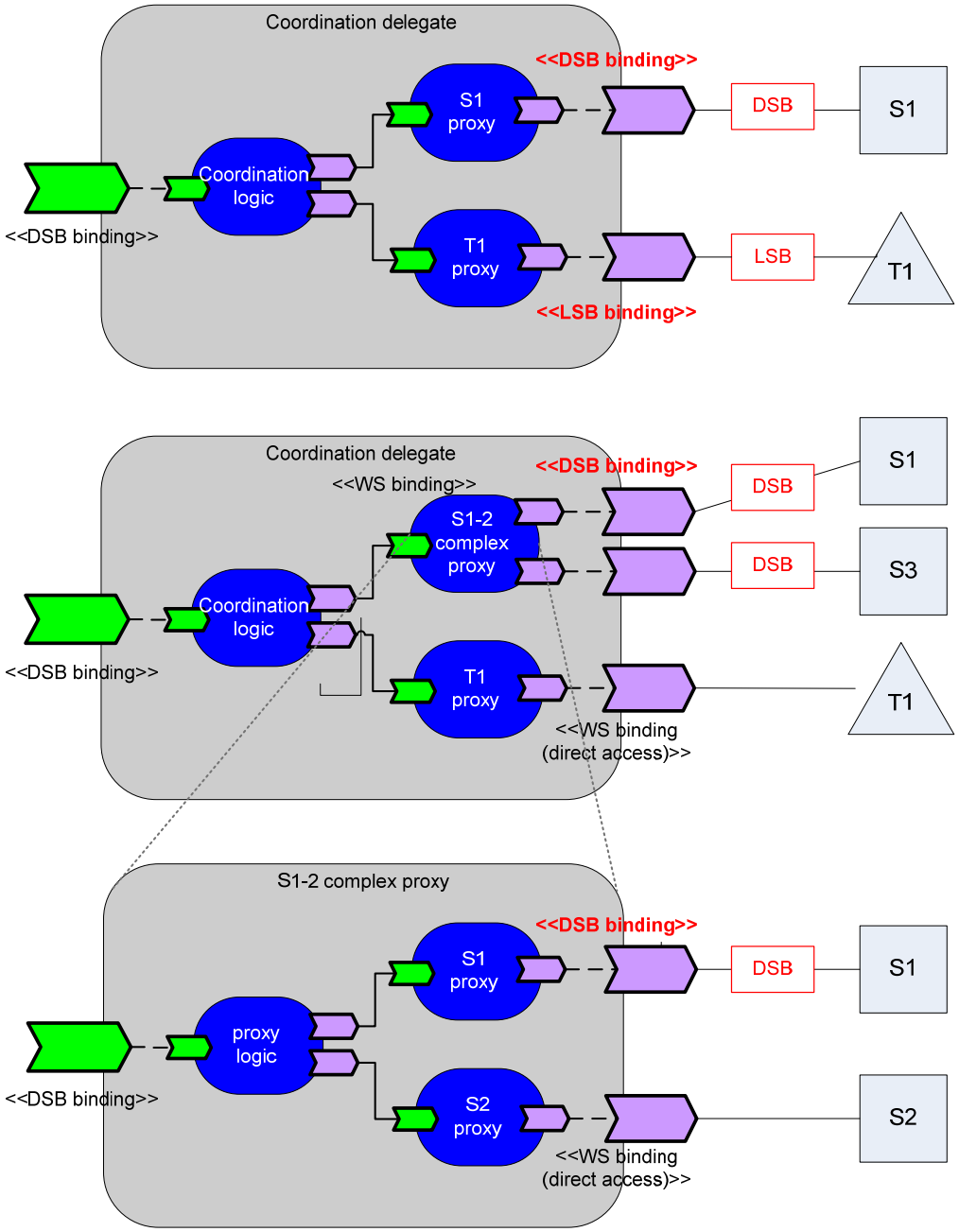

Figure 5-6. SCA mapping building blocks.

The implementation of coordination delegates and complex proxies will be based on the FraSCAti implementation of SCA.

Deployment & Execution

After their implementation, coordination delegates as well as simple and complex proxies, need to be deployed in the FraSCAti execution environment. To do so, they need to be packaged in a SCA *domain*. An SCA domain is a complete runtime configuration that defines the boundaries of all the deployed artefacts: components, but also wires and composites. The domain is described through:

- A **virtual domain-level composite** whose components are deployed and running.
- A set of installed **contributions** in order to execute components.
- A set of **logical services** to handle the virtual domain-level composite and the installed contributions.

All our components will be composed of a set of XML documents to configure them at runtime and other additional files required to enable their execution. These files can be packaged as a contribution in SCA, using one of the following interoperable formats: file system directory, OSGi bundle, compressed directory or JAR file.

Management of Non-Functional Choreography Requirements

As in the case of the ESB-based XSC, the management of non-functional choreography requirements is handled in close collaboration with the Governance framework developed in WP4 through the use of the business service monitoring capabilities of the Governance framework to monitor the good behaviour of a choreography.

To deal with this business service monitoring, we plan to investigate the use a FraSCAti extension to the SCA model that enables the runtime management of components and composites. This FraSCAti extension allows the dynamic binding of "technical" services (the business service monitoring in our case) to concrete SCA artifacts. The aforementioned binding is realized by means of interception techniques put in place by the FraSCAti platform.

Using interception mechanisms to integrate technical services with business code, allows keeping these concerns cleanly separated not only at design time, but also at runtime.

5.2. XSC for Thing-based Service Compositions

Differently from other choreographies described earlier in this chapter, most choreographies in the IoTS will require the coordination between numerous similar services. This is because any time an observation is made (or when an action is taken), it is necessarily tied to numerous sources of uncertainty that end up negatively influencing the result. The solution, then, is to add more information into the system by accessing additional sensors and actuators.

For this reason, choreographies in the IoTS will often require the interaction of a large number of *things*, many of which will be of the same general category (for instance "temperature sensor"), but perhaps located at different geographical coordinates, or produced by a different manufacturer. This process of combining numerous things to perform a sensing/actuation request is what we call a composition.

As described in Section 2.2.2, in the IoTS middleware, all incoming sensing/actuation requests are resolved by the Composition & Estimation component (Figure 5-1). C&E is in charge of coordinating the entire composition process, by relying heavily on semantic information stored in the XSD Knowledge Base (Figure 2-1). In addition, the C&E also interacts with the XSD Things Discovery protocol (Figure 2-1) to gain information regarding which things are available within the current network topology. Below, we describe each of these steps in further detail.

"Composition" consists of finding a dataflow graph that, given a description of the input parameters and the format of the desired output, connects the available *things* to produce the desired output from the parameters. To clarify, let us consider a brute-force implementation of the composition process, consisting of the following phases:

- **Expansion:** This step expands the initial query by replacing each term in the query with an equivalent expression, found by traversing the domain ontology. In this brute-force implementation, the final result of the expansion phase is a set of all possible combinations of service dataflows that answer the initial query.
- **Mapping:** This step takes all dataflows produced by the expansion step and maps them to the actual network topology. As such, mapping is necessarily performed by interacting closely with the XSD Things Discovery (Figure 2-1). This phase also interacts with our device ontology (present in the Knowledge Base) that models real world devices, to complement any information found to be missing during the discovery

process. The output of the brute-force mapping step is the set of all possible mappings of the input dataflows onto the network topology.

- **Optimal mapping selection:** Once all feasible dataflows have been mapped (in the mapping phase), C&E must choose one dataflow to enact. In this phase, therefore, a dataflow that is (in some predefined way) optimal is found and passed on to the execution block, below.
- **Execution:** Now that the best composition of services has been determined for the query, in the execution step the services are actually accessed (using the CHOReOS LSB showed in Figure 2-1) and the result is returned (or stored).

In an ultra-large-scale network with a large-scale Knowledge Base, seeking an optimal composition becomes an intractable problem. So, instead of pursuing an exact solution as was done in the brute-force case above, in our research we will pursue the idea of **approximately-optimal composition**, where the concepts of *expansion* and *mapping* are modified as follows:

1. **Smart expansion:** To avoid exhaustively calculating all possible equivalent sets of dataflows, only one of which will eventually be selected during the optimization phase, a much smarter approach to expansion is to instead produce a reduced set of good candidate dataflows. These candidates are the dataflows that have the highest likelihood of having a matching overlay in the network that satisfies a set of predefined constraints. An example constraint could be that the predicted execution time should fall within a certain acceptable interval.
2. **Probabilistic mapping:** Taking as input the set of candidate dataflows from the previous phase, the probabilistic mapping phase differs from regular mapping in that it does not attempt to find all possible mappings of the input dataflows into the network topology. Instead, this phase will randomly pick a small subset of all implementable mappings by making small, atomic queries to the probabilistic Discovery component. The result is a much reduced set of dataflow mappings that are computed in considerably less time and using (hopefully) orders of magnitudes less resources.

A consequence of the IoTS massive size and its unknown topology is that sometimes there will be no suitable device at the desired geographical location or, other times, the device has not collected/stored the data-point that is desired. However, often the missing data-points can be estimated with a very high degree of accuracy.

For instance, if an application would like to know the temperature at a location where no thermometer exists, then a sensing/actuation expert should be able to estimate the result using the values of the temperature readings in the surrounding area (for example, with a Kalman filter).

As such, to remove the dependency on the availability of a sensing/actuation expert, one of the goals of the C&E is to perform automated estimation. This is accomplished through the use of physical/statistical models that are provided a priori by field experts and made available in the Knowledge Base. Then, when an application makes a request for a data-point that is unavailable the network, the middleware can simply apply the provided models onto the time series of measurements from a set of sensors, and therefore estimate the most likely true value of the data at the desired spatiotemporal point. This process takes place in three steps:

- **Model discovery:** Look in the Knowledge Base for models related to the desired devices.
- **Optimal model selection:** Pick the most appropriate models based on a few parameters and a cost function (also specified in the ontology).

- **Estimation execution:** Apply the models to the existing historical data from sensors, using as input parameters the sensor and deployment metadata. This will be done using pre-developed engines for each model.

6. Cloud & Grid Computing

The CHOReOS middleware must be capable of providing the required runtime support to deploy, enact, monitor, and dynamically reconfigure large-scale choreographies. These choreographies might be large scale in one or more of the following dimensions: number of requests, users, participants, services, nodes, and communication among services. At this time in our research, we do not yet know the exact scale we will be able to achieve. But we expect, for instance, that the middleware should be scalable enough to accommodate a choreography with 1 thousand simultaneous users or with 100 different participants, or with 100 services for a given participant, or with thousands of messages exchanged per second. The exact scalability numbers will be known after the experiments carried out in the third year of the project.

To be able to accommodate such large magnitudes, the current solution provided by the state of the art of parallel and distributed computing relies on clusters of machines, often organized in federated groups across the Internet in geographically distributed locations. The CHOReOS middleware will benefit from two modern technologies developed within the last 10 years: Cloud and Grid Computing. Cloud Computing will be the default mechanism for providing scalability within CHOReOS while Grid Computing will be used in more specific cases in which parallel computation is required.

6.1. Cloud for CHOReOS

The allocation of Cloud machines for the execution of the choreographies is performed by the CHOReOS middleware in a way that is transparent to the choreography users, designers, and developers. The CHOReOS middleware uses the **Service Deployer** component to allocate new nodes from the **Node Pool Manager** and then execute new services in these nodes. In these nodes, CHOReOS will execute major choreography components (e.g., proxies, adapters, coordination delegates) for service access at runtime (Figure 2-1). To this end, the **Enactment Engine** will use the Node Pool Manager and the Service Deployer to set up the choreography environment and enable its execution.

To achieve the required level of scalability, the Node Pool Manager is able to allocate new nodes in multiple underlying execution platforms. As depicted in Figure 6-1, a CHOReOS node may be part of a Cloud Infrastructure as a Service (IaaS) platform; these can be provided by a public Cloud such as Amazon EC2, HP Open Cirrus, or a private Cloud, for example, executing the Open Nebula open source Cloud middleware. Both middleware and application services deployed on these Cloud nodes will rely on the CHOReOS XSB middleware for communication (Figure 2-1). After that moment, the CHOReOS monitoring service developed in WP4, in collaboration with WP3, will monitor the communication and the resource utilization in the Cloud nodes and provide up to date information to the runtime QoS and V&V enforcers that will guarantee that the choreography is executing correctly and that the SLAs are being met. If the system detects QoS violations, it might be needed to perform reallocation of services in the Cloud, requiring strict collaboration between components developed within WP3 and WP4.

Besides node provisioning, the CHOReOS Cloud middleware also provides storage capabilities. The Storage Service provides the proper infrastructure to store data in a scalable and robust way, facility that can be useful both to applications running on top of CHOReOS and to middleware components such as the AoSBM and the IoTS middleware elements. The CHOReOS Storage Service follows the Simple Storage Service (S3) API proposed by Amazon[14], which is becoming a *de facto* standard in the industry.

[14] See http://aws.amazon.com/s3

In the following subsections, we describe the interface provided by the Node Pool Manager to allocate new nodes, by the Service Deployer to instantiate new services on these nodes, by the Storage Service to provide access to storage in the Cloud and by the Enactment Engine to set up the execution of choreographies.

6.1.1. Node Pool Manager

The Node Pool Manager is composed of three main components: Controller, Configuration Management Engine, and Cloud Deployer as described in Figure 6-1.

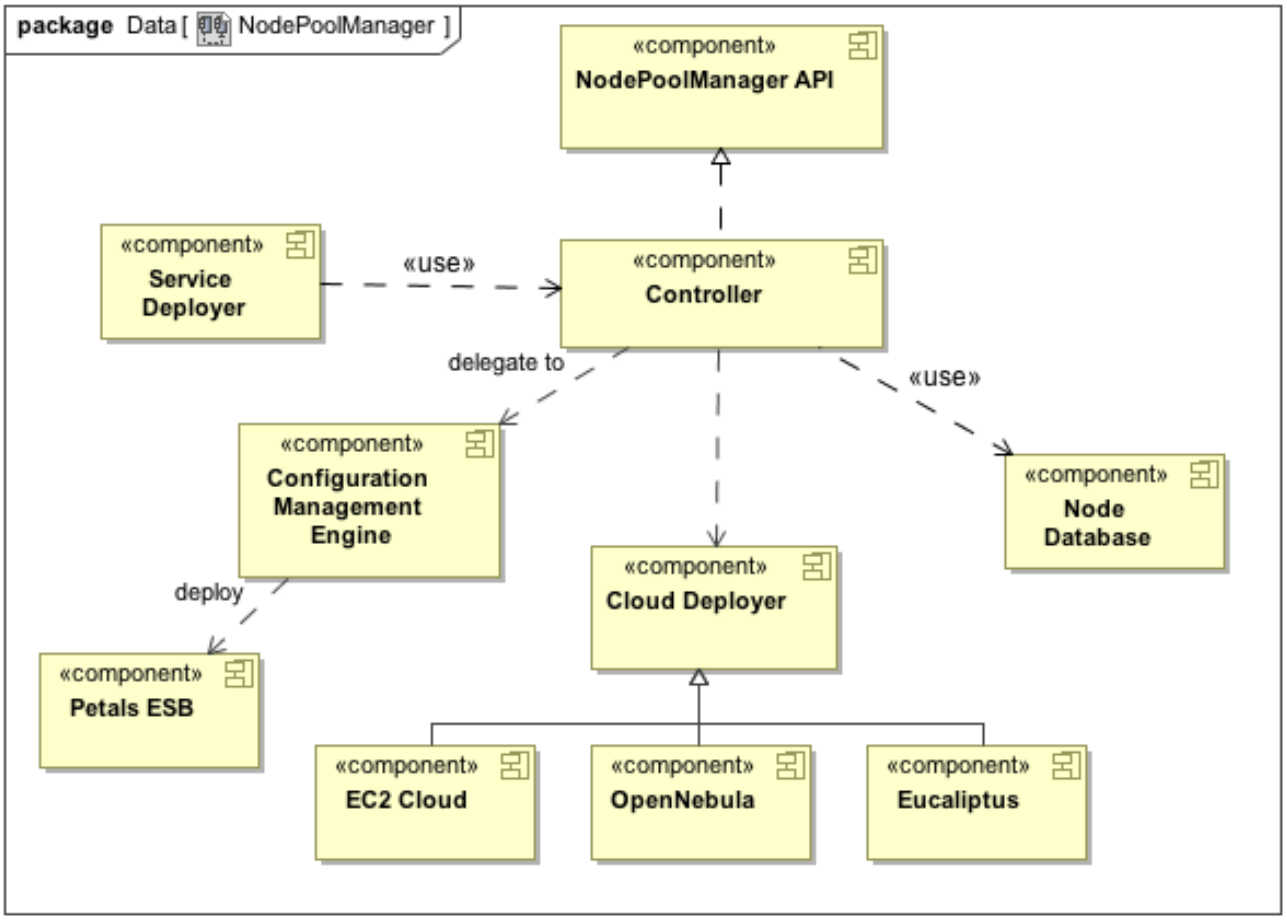

Figure 6-1. Node Pool Manager components.

The core component is the **Controller**, which provides the basic functionality of the Node Pool Manager; it implements a RESTful API that offers node management services to other Middleware components. The API can be implemented using a pure HTTP service or using Java JAX-RS, which basically converts a Java class to a REST service. The main entity created by this API is the CHOReOS Node. Each CHOReOS Node represents a logical machine created and managed by the CHOReOS Middleware. A CHOReOS Node can be created in any Cloud service (following the IaaS model) supported by the middleware. This will be transparent to other components that use the Node Pool Manager. Each CHOReOS Node also has a set of configurations. Each configuration is related to a node role and leads the middleware to install a set of software components in this node. The configuration management engine used by the Node Pool Manager provisions this installation.

The second component is the **Configuration Management Engine**, which is responsible for executing operating system level commands in CHOReOS Nodes, such as installing packages, starting and stopping services, creating and changing configuration files within the nodes. There are a few open source Configuration Management (CM) systems widely used such as Chef, Puppet, and CFEngine. After some studies, for the CHOReOS middleware, we chose to use Chef because it is easy to use, has an active community that keeps it up to

date, and is based on a simple Ruby-based domain-specific language, providing a good degree of flexibility and scalability. Nevertheless, the middleware architecture will be decoupled from specific details of the Chef tool, allowing us to replace it by another CM without affecting other components in the CHOReOS middleware, in case it does not satisfy our needs in the future. Chef has some core concepts that will be briefly described just for better understanding of how each node will be configured. Further details can be found in the Opscode Chef website (www.opscode.com/chef). In Chef, the system administrator or developer write Recipes that describe how a part of the server (such as Apache, MySQL, or Hadoop) must be configured. These recipes describe a series of Resources that should be in a particular state - packages that should be installed, services that should be running, or files that should be written. The CM makes sure each Resource is properly configured, and gives a safe, flexible, easily-repeatable mechanism for ensuring that the servers are always running exactly the way they are supposed to.

Every server in the infrastructure is a Node, which can have many Roles. The Role defines the purpose of that Node in the whole infrastructure. In a complex system, one can have nodes as web servers running Apache, others having MySQL databases, another with an LDAP server and a load balancer, and so on. Each Node can play one or more Roles.

After writing a Recipe, the administrator must put them in Cookbooks and store all this information in the Chef Server. After recipes are created, all that it is needed is to inform what recipes each Role will have. That is, all the recipes configured in a Role will be applied to each Node that has that specific Role assigned to it.

For the Node Pool Manager, every CHOReOS Node is a Node in the Chef environment. In addition, each Configuration applied to a Node is related to a Role in the Chef environment.

The third component of the CHOReOS Cloud/Grid infrastructure is the **Cloud Deployer**, which is responsible for the communication with the Cloud service providers. Since there can be multiple Cloud providers in a single choreography, this component will call each one of them properly, optimizing the resources and also granting redundancy in case of individual provider failure. This component will rely on the Open Cloud Computing Interface (OCCI) published by the Open Grid Forum (http://occi-wg.org), one of the first standards for Cloud computing. However, it is still possible to implement other patterns and use other specifications later if necessary.

The first version of the **Node Pool Manager API** has methods for creating nodes, deleting nodes, modifying existing node attributes, listing available nodes, showing existing node details, adding configuration to existing nodes, and removing configuration from nodes. These operations are the basic set that abstracts the cloud infrastructure to the rest of the Middleware components. The whole set of operations is described in Appendix D.1. As an example, we will describe here one of these operations in detail: the node creation operation.

HTTP Method	URI	Request Content Sample	Responses
POST	/nodes	<node> <cpus>2</cpus> <ram>1024</ram> <storage>100</storage> <so>Ubuntu 10.4</so> <zone>eu-west-1</zone> </node>	**201 CREATED** <node link="/nodes/{id}"/> **500 ERROR**

Table 6-1. CHOReOS Node creation.

A CHOReOS Node is created when there is a service call using the HTTP POST method on the /nodes URI. The content body for this POST is an XML request similar to the example given in Table 6-1. The Node attributes are explained in more detail in Table 6-2.

Name	Required	Type	Description
cpus	yes	Integer	Number of CPUs in a node.
ram	yes	Integer	Amount of RAM (in MB).
storage	yes	String	Amount of storage space (in GB).
hostname *	no	String	The node hostname.
IP*	no	String	The node IP address (if applicable).
so	yes	String	The operating system to be installed in the node. The SO is related to the bootstrap image that will be used to create the node. Nodes are, in essence, a SO installation plus a set of configurations.
zone	no	String	The zone attribute can be any String. Nodes with the same zone name will be prioritized to be created in the same cloud infrastructure.
* The IP and hostname attributes are not passed on node creation time, since they are set by the Cloud provider after the node is created. These attributes will be available later on the GET node details service.			

Table 6-2. CHOReOS Node attributes.

The CHOReOS Node creation sequence diagram (Figure 6-2) describes how information flows:

1. Another component (Service Deployer on this case) calls the node creation service;
2. The service delegates the real node creation to the Cloud Provider via the Cloud Deployer component;
3. Before the server is ready, the Node Pool Manager returns to the caller a message saying that this node is being created;
4. Once the Cloud provider finishes the server creation, the relevant information about the node is stored on the Node Pool Manager database;
5. The Node Pool Manager is asked to create a new configuration for that node;
6. It delegates this configuration to the Configuration Management tool;
7. It returns that this configuration is created;
8. The calling component asks for details about the Node;
9. The Node Pool Manager asks the Configuration Management tool for node details;
10. The Configuration Manager returns node information;
11. The Node Pool Manager joins its own information about nodes with the information provided by the Configuration Management tool and returns it to the caller.

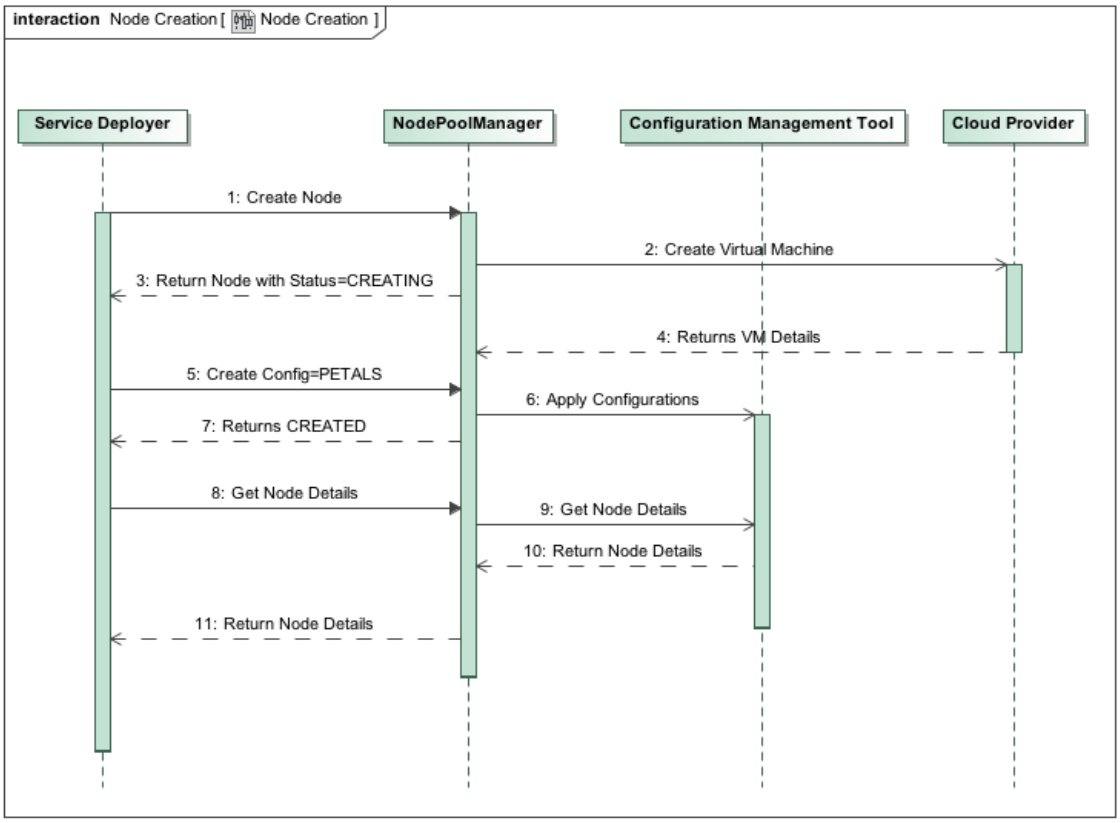

Figure 6-2. The CHOReOS Node creation sequence diagram.

6.1.2. Service Deployer

The Service Deployer is defined using a RESTful API. This component will deploy a given service to be later used by a choreography. Most deployment details are hidden from the client, such as in which machine to deploy the service and the number of instances. The client may provide hints as to the expected resource consumption for each instance and request non-mandatory increases or decreases in available processing power (which means modifying the number of instances according to an estimate).

The Deployer is capable of deploying the following kinds of elements:

- Web services
- BPEL orchestrations (not recursively, the needed services must be already available)

These elements are enough for running services choreographies. To deploy these kind of services, the Service Deployer needs the corresponding code in executable form (e.g., war files, BPEL specs, etc.).

The Service Deployer will also be able to remove (undeploy) these elements. The complete Service Deployer API can be found on Appendix D.2

6.1.3. Storage Service

Choreographies running within CHOReOS nodes may need to store and process large amounts of data. CHOReOS nodes responsible for running choreography business logic could store in their own file system the data required by these choreographies. If these data need to be accessed from other nodes, this would start to become complicated and error prone, since the nodes would need to be configured with a shared file system, or provide ways for other nodes to access their files. Scalability would also be compromised, as replicated services executed on multiple nodes would possibly need to share some data.

Another problem of storing data inside CHOReOS nodes is that some of them will not have disk space for storage. Since nodes can also be things (in the IoTS), these devices have limited storage space and must use external services to store large files (or also large amounts of small files).

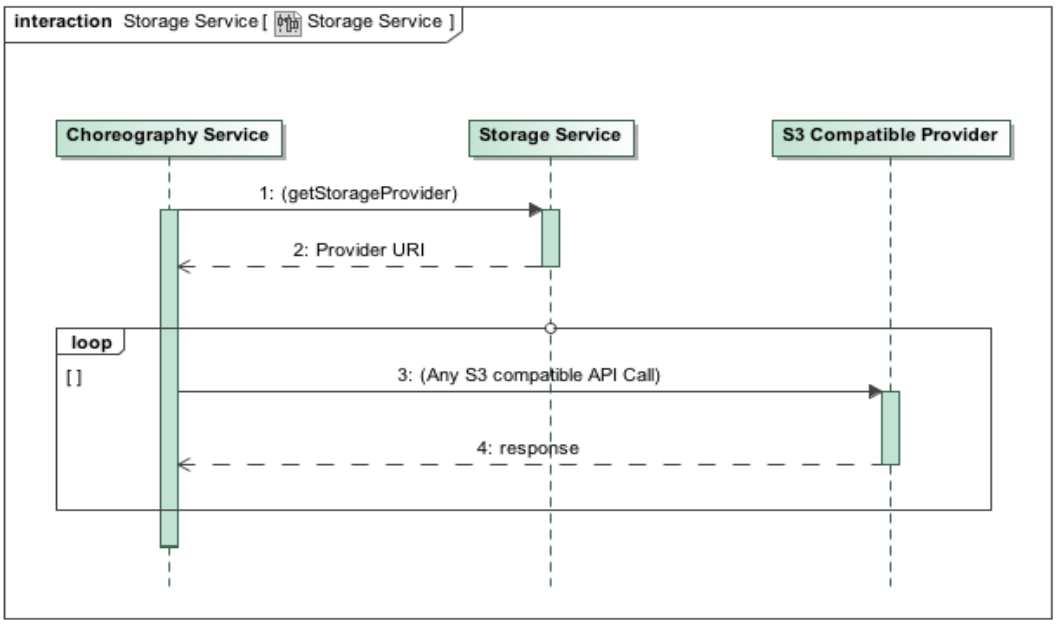

Figure 6-3. Storage service sequence diagram.

A more reasonable approach is using storage services in the Cloud. When a participant in the choreography needs to save or retrieve data, it just calls a public Storage Service API.

The CHOReOS Storage Service implements the Simple Storage Service (S3) API defined by Amazon. S3 is a Web service that enables any application to store data in the Cloud. One can then download the data or share it with other services. Amazon and Google Storage are examples of providers that already implement the S3 API, which is becoming a standard for storage services. The CHOReOS Storage Service usage is explained in Figure 6-3:

1. When any choreography service needs to store or retrieve data, it will call the Storage Service API and ask for a storage provider (`getStorageProvider()` call).
2. The Storage Service will infer the best provider for this service (based on the service node, its geographical location, etc.) and return the URI.
3. From this point, the service can start issuing as many calls as desired to the storage provider, using any S3 API call (documented in http://docs.amazonwebservices.com/AmazonS3)

Another usage example for storing data would be the relational database model. There are currently *ad hoc* solutions in Cloud infrastructures that provide Relational Database Services (e.g., Amazon RDS). Unfortunately, there is yet no established standard for a relational database service in the cloud industry; so, to provide a platform-independent cloud database service it would be necessary to create a common abstraction and mappings for each platform. Another simpler option, instead of using Cloud databases, since the Node Pool Manager can easily provision any kind of Node, would be to create new relational database nodes (e.g., using MySQL or PostgreSQL) or relational database clusters using existing configuration management open source recipes. During the second year of the project this topic will be further studied to define which approach CHOReOS will adopt.

6.1.4. Enactment Engine

The Enactment Engine will be the middleware component that will use the aforementioned services to set up the choreography environment and enable its execution on the Cloud. For that, it needs to receive as input, from the CHOReOS XSC, the following artifacts: (1) set of coordination delegates defining the interaction logic among the choreography services (e.g., in BPEL or jar files) and (2) services referenced by the coordination delegates, i.e., either already existing service URIs or binary code (e.g., war, jar, aar files) of the services that are not already deployed.

Provided the above input, the Enactment Engine will: (1) instantiate the coordination delegates using the Service Deployer; (2) deploy services that are not running using the Service Deployer; and (3) register services that are not already registered in the XSD.

The Service Deployer will make the services available through the DSB, abstracting details such as Petals Service Assembly specification.

The Enactment Engine will be used not only in production time, to enact the choreographies, but also at design and development time to validate and assess the choreography in an offline testing environment. In this latter case, not all choreography services may be available, thus mock services will need to be instantiated by the testing framework, with the help of the Enactment Engine.

A very preliminary version of the Enactment Engine API is presented in [D5.2] but this component will be developed in fact during the second year of the project.

6.2. Grid as a Service

Applications can be run in a Grid computing environment by interacting with the InteGrade Web Service Interface. Users will be able to start different types of applications (e.g., MPI, bag-of-tasks, and BSP), keep track of their progress, and receive the results. CHOReOS will also provide a similar service, following the map-reduce model, based on the Apache Hadoop infrastructure.

Grid Computing may be particularly interesting for end-user applications or middleware components that require a high degree of parallelism for computationally-intensive tasks. For example, the CHOReOS Grid could be used by a citizen journalism choreography that needs to convert large quantities of images and videos to different formats in a relatively short period (e.g., thousands of pictures and tens of videos per hour).

In this sense, the integration of Grid computing into CHOReOS does not intend to bring scientific advances to the area of Grid computing. Instead, the goal here is to incorporate a useful technology to be provisioned to CHOReOS applications and middleware.

6.2.1. CHOReOS InteGrade Interface

The Grid Computing Service in CHOReOS will provide the means for submitting a new application to the Grid (by providing its executable code) and for executing it in a large quantity of machines in parallel. Three programming models will be supported: bag-of-tasks, MPI, and BSP. The InteGrade grid middleware will be available as a Web Service through a REST API that allows users to upload applications and input data, specify constraints, check execution status and, finally, download output data (files, whole directories, simple text output or error messages).

As a usage example, consider the citizen journalism scenario. Thanks to technology, creating and disseminating news and information are not exclusive to professional journalists anymore. Inexpensive mobile phones with video camera are able to cover an increasingly growing area. Suppose somebody was in a singular event and has interesting videos to share. The user could post his video to a news portal and the portal will have to transcode that video as well as many others it receives from all its users into multiple formats that offer

more advantages to multiple types of viewing devices (e.g., more compatibility and less bandwidth usage). Using our REST API, portal developers had already uploaded an application (*POST /applications*) that, provided an input video file, produces a set of video files in the target formats. The same application will be used thousands of times with different input data. When the video is uploaded, another API operation is used (*POST /executions*), this time to send more input data to the application. It answers with an execution identifier, which the portal application uses to get its status periodically (*GET {identifier}*). When the status is "finished", the portal downloads the result (*GET {identifier}/result*) and makes it available to its viewers. The complete Grid Computing API can be found in Appendix D.3. If needed, a simple call-back mechanism can be implemented in the future so that the Grid infrastructure calls back a web service in the video portal to notify which videos have been converted and can be retrieved. Another possible improvement would be to make the grid middleware install the converted videos directly in the Cloud infrastructure and simply return a reference to where the videos are located.

6.2.2. CHOReOS Hadoop Interface

The CHOReOS middleware will also enable the execution of Hadoop applications based on the map-reduce processing model. Choreography services will be able to deploy and run Apache Hadoop applications on the cloud via the CHOReOS Hadoop Web Service.

The Hadoop Web Service will use almost the same interfaces described for the CHOReOS InteGrade interface. The only difference will be the way that data input and output for grid applications will be performed. In the Hadoop Web Service, the middleware will provide the *NameNode* configuration that is used to access the *Hadoop Distributed File System,* where the input and output files will be stored. Thus, the only modification with regard to the InteGrade interface will be the GET method described in Table 6-3.

HTTP Method	URI	Request Content	Responses
GET	/applications/ {application_id}/ namenode	-	**200 OK** <configuration> <property> <name>fs.default.name</name> <value>hdfs://localhost:54310/</value> </property> <property> <name>mapred.job.tracker</name> <value>localhost:54311</value> </property> </configuration> **404 NOT FOUND** **500 ERROR**

Table 6-3. Hadoop specific operation.

7. Conclusion

This deliverable introduced the basic features of the CHOReOS middleware architecture. Starting from the FI challenges for scalability, heterogeneity, mobility, awareness, and adaptation that have been extensively discussed in prior work done in WP1, we identified the aforementioned features to satisfy the requirements derived from the FI challenges. Specifically, the CHOReOS middleware contributions are summarized in the following:

- The CHOReOS XSD approach for the organization and discovery of services for both the IoBS and the IoTS domains. For the organization of services, XSD employs clustering mechanisms developed in WP2 for IoBS. XSD supports service discovery through an extensible set of service discovery protocols that range from legacy protocols to protocols dedicated to IoTS. Concerning the latter, we introduced the concept of probabilistic discovery that is encapsulated in the XSD.
- The CHOReOS XSB approach to the FI requirements for service access. Building upon a unified set of interaction primitives that constitute the Generic Application (GA) connector model developed in WP1 and the service bus paradigm, the XSB facilitates the integration of multiple interaction protocols that range from protocols suitable for the interaction with business services to protocols suitable for the interaction with Things-based services. Based on the same model, the XSB enables the adaptation from one protocol to another. The aforementioned integration and adaptation facilities are supported by the two CHOReOS service buses, namely the DSB and the LSB, which target the IoBS and the IoTS domains, respectively.
- The CHOReOS XSC solutions for the execution of FI service compositions. The proposed solutions cover both the IoBS and the IoTS domains. Regarding the former domain, the XSC provides approaches that facilitate the execution of business service choreographies defined according to the CHOReOS development process, developed in WP2. Concerning the latter domain, we introduced the concept of approximate service composition to deal with the massive composition of things.
- Finally, the CHOReOS Cloud & Grid middleware that supports the overall middleware and the choreographies that rely on it, via a unified API that provides access to multiple cloud infrastructures (e.g., Amazon EC2, HP Open Cirrus, private clouds).

Based on the aforementioned features that characterize the architecture of the CHOReOS middleware, we proceed with the first prototype implementation of the middleware, which shall serve for further refining, extending, and concretizing the proposed ideas.

References

[ABBC03] A. Arasu , B. Babcock , S. Babu , J. Cieslewicz , K. Ito , R. Motwani , U. Srivastava, J. Widom. STREAM - the Stanford Data Stream Management System. In Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems 2003

[AHS07] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for Data Processing in Large-scale Interconnected Sensor Networks. In Proceedings of the International Conference on Mobile Data Management, 2007, pages 198–205.

[CASAGRAS09] CASAGRAS group. RFID and the Inclusive Model for the Internet of Things. http://www.rfidglobal.eu/userfiles/documents/FinalReport.pdf, 2009.

[CCDF03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. Shah. Telegraphcq: Continuous Data Flow Processing. In Proceedings of the International Conference on Management of Data (SIGMOD), 2003.

[CCEG03] M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora - A Data Stream Management System. In Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD) 2003

[CMMP07] P. Costa, L. Mottola, A. Murphy, and G. Picco. Programming Wireless Sensor Networks with the Teeny Lime Middleware. In Proceedings of the International Middleware Conference, 2007.

[CMNG08] D. Cook, J. Mejino JR. M.Neal and J. Gennari. Bridging Biological Ontologies and Biosimulation: the ontology of physics for biology. In Proceedings of the AMIA Annual Symposium, vol. 2008, American Medical Informatics Association, pages 136–140, 2008.

[COBIS07] CoBIs project. Cobis final project report deliverable 104 v 2.0. Tech. Rep., 2007. [Online]. Available: http://www.cobis-online.de/files/ Deliverable D104V2.pdf

[CONN-D1.2] CONNECT Project Team. D1.2: Intermediate CONNECT Architecture. Technical Report D1.2, CONNECT ICT FET IP Project, http://connect-forever.eu/, February 2011.

[CRI10] M. Caporuscio, P.-G. Raverdy, and V. Issarny. Ubisoap: A Service Oriented Middleware for Ubiquitous Networking. IEEE Transactions on Services Computing, vol. 99 (PrePrints), 2010.

[D1.2] CHOReOS Project Team. D1.2: Choreos Perspective on the Future Internet and Initial Conceptual Model. www.choreos.eu, April 2011.

[D1.3] CHOReOS Project Team. D1.3: Initial Architectural Style for CHOReOS Choreographies. www.choreos.eu, Oct. 2011.

[D2.1] CHOReOS Project Team. D2.1: CHOReOS Dynamic Development Model Definition. www.choreos.eu, Oct. 2011.

[D3.1-Comp] CHOReOS Project Team. D3.1 Companion Deliverable: Survey on DPWS. www.choreos.eu, September 2011.

[D4.1] CHOReOS Project Team. D4.1: Governance V&V Policies and Rules. www.choreos.eu, Oct. 2011.

[D5.2] CHOReOS Project Team. D5.2: Specification of the CHOReOS IDRE.

www.choreos.eu, Oct. 2011.

[D6.1] CHOReOS Project Team. D6.1: Requirements and Scenarios for the Passenger Friendly Airport. www.choreos.eu, Oct. 2011.

[Daras09] Daras P. et Al., Why Do We Need a Content-Centric FI ? Proposals Towards Content-Centric Internet Architectures. Information Society and Media Journal, 2009.

[DGV09] S. Duquennoy, G. Grimaud, and J. Vandewalle. The Web of Things: interconnecting devices with high usability and performance. In Proceedings of the IEEE International Conference on Embedded Software and Systems (ICESS), 2009, pages 323-330.

[DLW08] R. Dickerson, J. Lu, J. Lu, and K. Whitehouse. Stream Feeds - An Abstraction for the World Wide Sensor Web. The Internet of Things, pages 360-375, 2008.

[DNJC10] A. Devaraju, H. Neuhaus, K. Janowicz, and M. Compton. Combining Process and Sensor Ontologies to Support Geo-sensor Data Retrieval. In Proceedings og the 6th International Conference on Geographic Information Science, pages 1–5, 2010.

[ELS07] M. Eid, R. Liscano, A. E. Saddik. A Universal Ontology for Sensor Networks Data. In Proceedings of the IEEE International Conference on Computational Intelligence for Measurement Systems and Applications, pages 27-29, 2007

[ERA10] M. Eisenhauer, P. Rosengren, and P. Antolin. Hydra: A Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence systems. The Internet of Things, pages 367–373, 2010.

[ETP09] FI, The Cross-ETP Vision Document, 2009, Available at http://www.future-internet.eu/fileadmin/documents/reports/Cross-ETPs_FI_Vision_Document_v1_0.pdf.

[FRL09] C. Fok, G. Roman, and C. Lu. Agilla: A Mobile Agent Middleware for Self-adaptive Wireless Sensor Networks. ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol 4, no. 3, pages 1-26, 2009.

[GCG94] N. Guarino, M. Carrara, and P. Giaretta. An Ontology of Meta-level Categories. In Proceedings of the 4th International Conference Principles of Knowledge Representation and Reasoning, 1994, pages 270–280.

[GKKN03] P. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An Architecture for a Worldwide Sensor Web. IEEE Pervasive Computing, pages 22-33, 2003.

[GPJM08] M. Gomez, A. Preece, M. Johnson, G. De Mel, W. Vasconcelos, C. Gibson, A. BarNoy, K. Borowiecki, T. La Porta, D. Pizzocaro, et al. An Ontology-centric Approach to Sensor-Mission Assignment. Knowledge Engineering: Practice and Patterns, pages 347–363, 2008.

[GTKS10] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-demand Provisioning of Web Services. IEEE transactions on Services Computing, vol. 3, no. 3, pages 223–235, 2010.

[GTMW10] D. Guinard, V. Trifa, F. Mattern and E. Wilde. From the Internet of Things to the Web of Things: Resources Oriented Architecture and Best Practices. Architecting the Internet of Things, Springer, 2010

[GTW10] D. Guinard, V. Trifa, and E. Wilde. A Resource Oriented Architecture for the Web of Things. In Internet of Things (IOT), 2010, pages 1-8. IEEE, 2010.

[GWMA06] J. Girao, D. Westhoff, E. Mykletun, and T. Araki. TinyPEDS: Persistent Encrypted Aata Storage in Asynchronous Wireless Sensor Networks. Elsevier

Ad-hoc Networks Journal, 2006.

[HBD09] A. Hornsby, P. Belimpasakis, and I. Defee. XMPP-based Wireless Sensor Network and its Integration into the Extended Home Environment. In Proceedings of the IEEE 13th International Symposium on Consumer Electronics, pages 794-797, 2009.

[HHLL03] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In Proceedings of the 29th International Conference on Very Large Data Bases (VLDB) - Volume 29, pages 321-332, 2003.

[HLBT10] J. Honkola, H. Laine, R. Brown, and O. Tyrkko. Smart-M3 Information Sharing Platform. In Proceedings of the IEEE Symposium on Computers and Communications (ISCC), 2010, pages 1041–1046.

[HTS08] U. Hunkeler, H. Truong, and A. Stanford-Clark. Mqtt-s a Publish/Subscribe Protocol for Wireless Sensor Networks. In Proceedings of the 3rd IEEE International Conference on Communication Systems Software and Middleware and Workshops, 2008., pages 791-798.

[IGHZ11] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadis, M. Autili, M. A. Gerosa and A. Ben Hamida. Service-oriented Middleware for the FI: state of the art and research directions. Journal of Internet Services and Applications (JISA), vol. 2, no. 1, pages 23-45, 2011, Springer.

[JLKY08] J. Jung, S. Lee, N. Kim, H. Yoon. Efficient Service Discovery Mechanism for Wireless Sensor Networks. Computer Communications, pages 3292–3298, Elseveir, 2008.

[Kay04] M. Kay. XSLT 2.0 Programmer's Reference. Wiley Pub. (2004).

[KGB10] K. Krizanovic, Z. Galic and M. Baranovic. Spatio-temporal Data Streams: An Approach to Managing Moving Objects. In Proceedings of the 33rd IEEE International MIPRO Convention, pages 744-749, IEEE, 2010.

[KKKN08] A. Katasonov, O. Kaykova, O. Khriyenko, S. Nikitin, and V. Terziyan. Smart Semantic Middleware for the Internet of Things. In Proceedings of the 5th International Conference on Informatics in Control, Automation and Robotics, pages 11–15, 2008

[KKLK08] E. Kang,M. Kim, E. Lee, and U. Kim. DHT-based Mobile Service Discovery Protocol for Mobile Ad Hoc Networks. Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues, pages 610-619, Springer, 2008.

[KNLZ07] A. Kansal, S. Nath, J. Liu, and F. Zhao. Senseweb: An Infrastructure for Shared Sensing. IEEE multimedia, vol. 14, no. 4, pages 8-13, 2007.

[Kuhn09] W. Kuhn. A Functional Ontology of Observation and Measurement. GeoSpatial Semantics, pages 26–43, 2009.

[LDMD11] T. S. Lopez, R. Damith, H. Mark, M. Duncan. Adding Sense to the Internet of Things. Personal and Ubiquitous Computing, Springer, 2011, pages 1-18.

[LZ05] Liu J. and Zhao F. Towards Semantic Services for Sensor-Rich Information Systems. In Proceedings of the 2nd IEEE/CreateNet International Workshop on Broadband Advanced Sensor Networks, 2005.

[MDMV09] D. Massaguer, M. Diallo, S. Mehrotra, and N. Venkatasubramanian. Middleware for Pervasive Spaces: Balancing Privacy and Utility. In Proceedings of the 10th ACM/IFIP/USENIX International Middleware Conference. LNCS, 2009, vol. 5896, pages 247–267.

[MFHH05] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. ACM Transactions on Database Systems (TODS), vol. 30, no. 1, pages 122-173, 2005.

[MVOP08] J. Macedo, C. Vangenot, W. Othman, N. Pelekis, E. Frentzos, B. Kuijpers, I. Ntoutsi, S. Spaccapietra and Y. Theodoridis. Mobility, Data Mining and Privacy – Geographic Knowledge Discovery. Trajectory Data Models, pages 123-150. Springer, 2008.

[NC09] H. Neuhaus and M. Compton. The Semantic Sensor Network Ontology. In AGILE Workshop on Challenges in Geospatial Data Harmonisation, pages 1–33, 2009.

[NLZ07] S. Nath, J. Liu, and F. Zhao. Sensormap for Wide-area Sensor Webs. Computer, vol. 40, no. 7, pages 90-93, 2007.

[NSM10] T. Narock, A. Szabo, and J. Merka. Using Semantics to Extend the Space Physics Data Environment. Computers and Geosciences, pages 791 – 797, 2009.

[PA06] S. L. Pfleeger and J. Atlee. Software Engineering: Theory and Practice, 3rd Edition, 2006, Prentice Hall.

[PBEV09] M.Presser, P.Barnaghi, M.Eurich, and C.Villalonga. The SENSEI Project: Integrating the Physical World with the Digital World of the Network of the Future. IEEE Communications Magazine, vol. 47, no. 4, pages 1–4, 2009.

[Pereira08] J. Pereira. From Autonomous to Cooperative Distributed Monitoring and Control: Towards the Internet of Smart Things. In Proceedings of the ERCIM Workshop on eMobility, 2008.

[PH07] M. P. Papazoglou, W. J. Heuvel. Service Oriented Architectures: Approaches, Technologies and Research Issues. VLDB Journal vol. 16, 2007, pages 389 – 415.

[PH09] D. Le-Phuoc and M. Hauswirth.. Linked Open Data in Sensor Data Mashups. Semantic Sensor Networks, pages 1–16, 2009.

[PKGZ08] B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny Web Services for Sensor Device Interoperability. In Proceedings of the IEEE International Conference on Information Processing in Sensor Networks, pages 567-568., 2008.

[PTDL07] Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. IEEE Computer vol. 40, no. 11, 2007.

[RG07] S. Rooney and L. Garces-Erice. Messo & Preso Practical Sensor-Network Messaging Protocols. In Proceedings of the 4th IEEE European Conference on Universal Multiservice Networks (ECUMN), pages 364-376, 2007.

[RICL07] P. Raverdy, V. Issarny, R. Chibout, and A. de La Chapelle. A Multi-Protocol Approach to Service Discovery and Access in Pervasive Environments. In Proceedings of the 3rd IEEE Annual International Conference on Mobile and Ubiquitous Systems-Workshops, 2006, pages 1–9, 2007.

[RKSE03] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric Storage in Sensornets with ght, a Geographic Hash Table. Mobile Networks and Applications, vol. 8, pages 427–442, 2003.

[SMFD09] Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J-B.: Reconfigurable SCA Applications with the FraSCAti Platform. In Proceedings of the 6th IEEE International Conference on Service Computing (SCC) pages

268-275, 2009.

[SPDM08] S. Spaccapietra, C. Parent, M. L. Damiani, J. A. de Macedo, F. Porto and C.Vangenot. A Conceptual View on Trajectories. Data & Knowledge Engineering, 2008.

[Stirbu08] V. Stirbu.Towards a RESTful Plug and Play Experience in the Web of Things. In Proceedings of the IEEE International Conference on Semantic Computing, pages 512-517, 2008.

[SZ09] P. Stuckmann, R. Zimmermann. European research on FI Design. IEEE Wireless Communication, pages 14-22, 2009.

[TGDK10] V. Trifa, D. Guinard, V. Davidovski, A. Kamilaris, and I. Delchev. Web Messaging for Open and Scalable Distributed Sensing Applications. In Proceedings of the International Conference on Web Engineering (ICWE), 2010.

[THVG11] T. Teixeira, S. Hachem, V. Issarny., and N. Georgantas. Service Oriented Middleware for the Internet of Things: A Perspective. In Proceedings of Servicewave, to appear.

[TSFH09] I. Toma, E. Simperl, A. Filipowska, G. Hench, J. Domingue. Semantics-Driven Interoperability on the FI. In Proceedings of the International Conference on Semantic Computing, 2009.

[WSBC04] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a Neighborhood Abstraction for Sensor Networks. In Proceedings of the 2nd ACM International Conference on Mobile Systems, Applications, and Services, pages 99-110, 2004.

[WXCJ98] O. Wolfson, B. Xu, S. Chamberlain and L. Jiang. Moving Objects Databases: Issues and Solutions. In Proceedings of the 10th International Conference on Scientific and Statistical Database Management, p: 111-122, 1998.

[WZL06] K. Whitehouse, F. Zhao and J. Liu. Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. Wireless Sensor Networks, pages 5–20, 2006.

[ZH08] W. Zhang and K. Hansen. Semantic Web-based Self-management for a Pervasive Service Middleware. In Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems, pages. 245–254, 2008.

[ZH09] W. Zhang and K. Hansen. An Evaluation of the NSGA-II and MOCell Genetic Algorithms for Self-management Planning in a Pervasive Service Middleware. In Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems, pages 192–201, 2009.

[ZPAG10] A. Zafeiropoulos, G. Panagiotis, L.Athanassios, M. Gregoris, M. Nikolas. NEURON: Enabling Autonomicity in Wireless Sensor Networks. Sensors, pages 5233-5262, 2010

Appendix A – Middleware for the IoTS

The Internet of Things is characterized by the integration of large numbers of real-world objects (or "things") onto the internet, with the aim of turning high-level interactions with the physical world into a matter as simple as is interacting with the virtual world today. As such, two devices that will play a key role in the IoT are sensors and actuators (S&A).

In fact, such devices are already seeing widespread adoption in the highly-localized systems within our cars, mobile phones, laptops, home appliances, etc. In their current incarnation, however, sensors and actuators are used for little more than low-level inferences and basic services. This is partly due to their highly specialized domains (signal processing, estimation theory, robotics, etc.), which demand application programmers to also be domain experts, and partly due to a glaring lack of interconnectivity between all the different devices.

To be truly useful, sensors and actuators must be ubiquitous rather than constrained to an area around a small set of personal devices, such as a mobile phone or a car. This translates to having a network with a massive number of "things", spread over a large area — possibly the entire world. As the number of sensors and actuators in a network grows to the millions, however, several challenges arise. In this appendix, we present our Middleware for the Internet of Thing-based services, which addresses the following challenges of the IoTS:

- **Scale**: When performing a sensing or actuation task that pertains to millions of devices, it is often infeasible to coordinate every one of the required devices due to constraints such as time, memory, processing power, and energy consumption. To put this into perspective, consider the simple case of an application that requires to know the average air temperature on the city of Paris at this very moment. The answer to this query can be "easily" found by calculating the mean value of the set of temperature readings all the thermometer-carrying devices in the region. However, if there are millions[15] of such devices in Paris, then the information being accessed in this query can quickly grow unmanageable. Thus even a simple-looking query such as this often leads to unattainable results when the scale is factored in. (In this example, a better solution is to approximate the average temperature within an acceptable level of precision by using the well-known equations for the sampling error of the mean.)

→ As a result, any realistic middleware for the IoTS must be able to manage on-the-fly **approximations** to sensing/actuation tasks. For this, we introduce the concept of **probabilistic discovery** to find the subset of devices that can provide a useful approximation when an exact result is not attainable.

- **Deep heterogeneity**: An important aspect of the IoTS that is usually not emphasized enough is that services representing *things* are much more heterogeneous than typical services on the current Internet. For one, due to cost considerations, new sensing/actuating hardware *will often not replace* older generations in already-deployed networks --- rather, different generations of devices will operate alongside one another. Likewise, it is probable that the future Internet will be composed of numerous sensor/actuator networks deployed by distinct entities, for distinct reasons. In all of these cases, these networks are bound to contain devices from an assortment of vendors, with highly varying sensing/actuating characteristics, such as error distributions, sampling rates, spatial resolution, and so on. All of these parameters (including functional and non-functional properties) lead to a *deep heterogeneity* that makes S&A networks extremely hard to work with, even for experts. And as networks increase in size, delegating these types of coordination tasks to humans will simply not

[15] This can happen if all cell-phones are instrumented with thermometers, for instance.

be feasible. In such a dynamic environment, with so many unknowns, it is clear that fully automated methods for high level inference will become a necessity.

→ *Our middleware* aims *to solve this challenge by making extensive use of* ***metadata*** *and* ***semantics (ontologies)*** *as building blocks that allow us to handle this heterogeneity.*

- **Unknown topology**: Much like the existing Internet, one of the IoTS's main characteristics is the fact that its topology is both unknown and dynamic. As a consequence, applications will often end up depending on services which are not actually available from any single preexisting component of the network at that given time. For instance, if an application would like to obtain the value of the wind-chill factor at a certain location, it may happen that the network does not have a wind-chill sensor in that exact neighborhood. However, if instead the network does have temperature and wind speed sensors (i.e. anemometer), then a field expert could easily obtain the desired information through the composition of the temperature and wind speed readings using the well-known wind-chill equation. This is possible because the *function* of a wind-chill sensor is equivalent to the *function* provided by the thermometer/anemometer combination put together by the expert. The question is, then: can an IoTS middleware perform these types of functional substitutions on its own without supervision? How can this type of service composition be performed in an optimal manner when the network is massive in scale, with unknown topology?

→ *Our middleware will solve this using* ***service composition****, which is able to use sensor metadata and semantics to create new services by connecting existing pieces.*

- **Unknown data-point availability:** A second consequence of the unknown topology is that sometimes there will be no suitable device at the desired geographical location or, other times, the device has not collected/stored the data-point that is desired. However, oftentimes the missing data-points can be estimated with a very high degree of accuracy. For instance, if an application would like to know the temperature at a location where no thermometer exists, then an expert should be able to estimate the result using the values of the temperature readings in the surrounding area (for example, with a Kalman filter). Or when the application requires access to the location of a car at some time t_1, but only the locations at time t_0 and t_2 are known ($t_0 < t_1 < t_2$), then a user versed in Newton's laws of motion should be able to calculate the midpoint-speed t_1. But how can these estimations take place in an automated fashion, without the need for human intervention?

→ *Our middleware will make use of* ***physical and statistical models*** *(stored in an domain-specific ontology) to achieve* ***automated estimation*** *of missing data and* ***indirect sensing****.*

- **Incomplete or inaccurate metadata**: The solution to many of the challenges above likely lies on the extensive use of metadata. However, since much of this metadata must be manually entered by a human operator at installation time, in a massive network this will surely result in a large amount of incomplete/inaccurate information due to human error. In addition, some of this information includes characteristics that change over time (e.g., calibration parameters). Therefore, even discounting human error, the state of the metadata in the network is bound to degrade until it no longer represents reality. In these scenarios, how can missing metadata be recovered? And how can existing information be monitored and updated when necessary?

→ *The previously-mentioned support for automated estimation in our middleware can also provide* ***auto-calibration*** *and* ***metadata-mining*** *procedures in order to constantly recheck the suitability of its calibration parameters. We do not address this at the moment, leaving it for future work.*

- **Conflict resolution**: Conflict resolution is an issue that arises mainly with actuators, but not so much with sensors. Conflicts arise, for instance, when multiple applications

attempt to actuate the same device in opposing ways, or when they would like to exert mutually-incompatible changes on the environment. For example, in a scenario where a smart building is able to adapt to people's personal temperature preferences, one person may want a choose a temperature of 17C while the other 25C. A human mediator would likely resolve this conflict using the average of the two temperatures, 21C. However, a much tougher example presents itself in the actuation of pan-tilt-zoom cameras: if one application requires the camera to turn left, and the other requires it to turn right, how can the network satisfy both applications — or at least gracefully degrade their quality of service?

→ *This issue is not yet considered in our middleware, since its initial focus is on sensors rather than actuators. However, this is a clear direction for future work and should not be kept too far out of sight.*

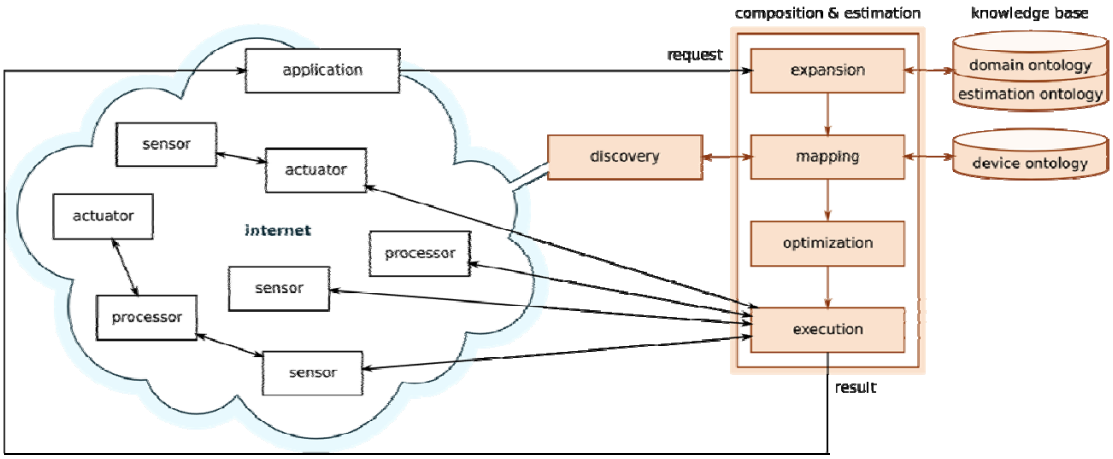

Figure A-1. Architecture of the CHOReOS IoTS Middleware.

Following, in Section A.1 we consider related work from the current literature and discuss the how our approach differs from them. In Section A.2 we provide an overview of the IoTS middleware, while in Sections A.3 to A.6 we discuss in more detail the elements of the proposed middleware. In Section A.7 we present our first prototype. Finally, in Section A.8 we summarize the discussion on the proposed middleware for IoTS.

A.1. Related Work

Most solutions towards an Internet of Things identify common challenges that should be addressed in order to successfully integrate devices with services in the Future Internet. Challenges include mobility of nodes, the dynamic nature of the network, heterogeneity of devices and data, availability of nodes and scalability (in terms of the number of nodes, users, data streams, etc.). A popular solution to these issues is the adoption of a middleware architecture where devices are abstracted as services. Those solutions can be divided in two categories: those that abstract devices as services (such as in HYDRA [ZH09, ERA10, ZH08], SENSEI [PBEV09], SOCRADES [GTKS10], and COBIS [COBIS07]), and those that devote attention to data/information abstractions and their integrations with services (among which are SOFIA1 [HLBT10], SATware [MDMV09], and Global Sensor Networks GSN [AHS07]).

A common thread throughout all these solutions is that they handle the challenge of **unknown topology** through the use of *traditional* service/device discovery techniques of the existing Internet, ubiquitous environments and Wireless Sensor & Actuator networks (e.g. SOCRADES uses DPWS, COBIS and HYDRA use UPnP). They focus on supporting discovery for devices hosting Web services (as done in SOCRADES) and/or RESTful services (as done in Stribu08, GTMW10, SENSEI). COBIS supports WS-Discovery but it provides its own language (COBIL, Collaborative Business Item Language) for service description, where service functions and keywords are annotated with a verbal description.

SOCRADES, for instance, supports both discovery techniques and goes a step further by providing an expansion approach, where additional information about services can be extracted from readily-available public sources such as Web search engines or encyclopedias. A similar approach was adopted in [LDMD11] where online information services such as EPC information services can be queried to provide additional information about sensors and actuators. Another important contribution by SOCRADES is the support for context aware discovery where services are selected based on the context they satisfy.

A common point of agreement in the state of the art is in the use of semantic technologies [ZH09, ERA10, ZH08, HLBT10, PBEV09]. These technologies have two main benefits: overcome **heterogeneity** challenges by abstracting device/service/information low-level details; increase the visibility of and the criteria by which services/devices can be discovered through the annotations of service and device information with metadata. Indeed it is standard practice to use ontologies to model sensors, their domains, and sensor data repositories [ELS07, LZ05, ERA10]. Some projects even go a step further and also include context information [PBEV09], or service descriptions [ZH09, ERA10, ZH08]. Semantic technologies are also used for service composition, as many projects support the concept of virtual/semantic sensors (for instance, in HYDRA, GSN and SATware), i.e. entities that abstract several aggregated physical devices under a single service. A different implementation of a similar idea, though, is provided in the SATware project: in their work, virtual sensors actually correspond to transformations applied to a set of raw sensor streams to produce another semantically meaningful stream. Although it can be said that the concept of virtual sensors is a sort of service composition, one must be careful to point out that this composition is *not fully dynamic*, in that the services are first specified at *design time*, and only then are they dynamically mapped onto the network at run time. In contrast, a much more flexible type of composition is to perform both operations at run time, through the help of small predefined composition building blocks as supported by the SENSEI and SOCRADES projects.

Regarding **scalability**, most IoT projects address this challenge by pursuing modifications in the underlying network topology. At times, this is done by adopting fully-distributed infrastructures (such as in COBIS and SOFIA), and at other times through an architecture of peer-to-peer clusters (e.g., GSN). In our view, however, while these approaches work well for the existing Internet (where traffic is made up of a relatively small amount of service interactions) they are not fit for the complex weave of interactions that will be commonplace in the Internet of Things. Some solutions focus on scaling the discovery process by proposing the use of DHT-based discovery [ZPAG10, KKLK08, JLKY08]. However, in our view these approaches fall short of fully addressing the scale of the IoT, which we estimate will consist of billions of nodes. For instance, authors in [JLKY08] assume topology-aware networks, which is something that is clearly not achievable at these projected scales. Meanwhile other solutions do not offer a convincing argument when it comes to evaluating their systems through either experiments or simulations. For example, in [KKLK08], service discovery is simulated in a 100-node mobile ad-hoc network, a number too small to represent the anticipated size of IoT. In our view, DHT-based routing on its own is not sufficient to address the problems of the Internet of Things, where a large number of requests will require intricate coordination among thousands of things, services and information they produce. In such an environment, the number of packets transmitted in the network will grow strongly nonlinearly as the number of available services and devices increases. As a result, routing tables will consist of either an ultra large number of nodes (each hosting a small portion of information), or a smaller number of nodes holding a huge amount of information each. Performing even a simple service discovery query in these scenarios may result in unacceptable response delays due to the propagation time of the request to large number of nodes in the routing table, not to mention the processing and memory overhead on each node itself.

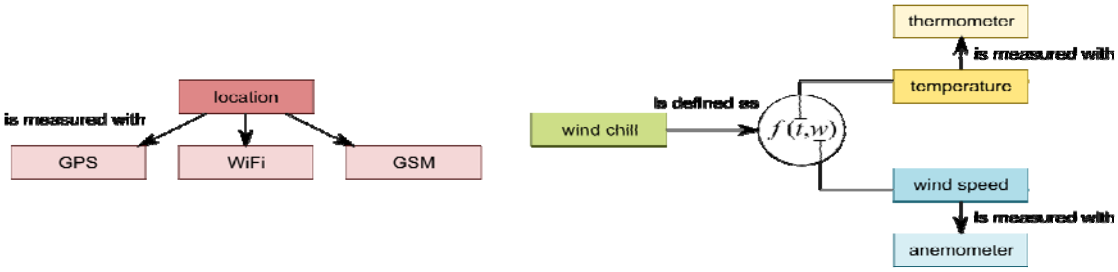

Figure A-2. Examples of types of relations that exist in our semantic Knowledge Base.

Finally, among the aforementioned projects, to the best of our knowledge, *none* considers the challenges of **data-point availability**, **inaccurate metadata**, and **conflict resolution**. To address such issues, in our proposed work, we plan to go a step further by considering situations where the system must smartly estimate sensor readings at time instants and geographic locations for which there is no appropriate device (this is what we call *estimation* throughout the text), and where the system's massive scale is handled through the widespread use of approximations. Our middleware extends ontologies with physical and statistical models that can be used to "fill in the blanks" where appropriate. By supporting concepts from signal processing, estimation theory, machine learning, and data mining, we provide a framework through which experts in those fields can program their latest estimation models that can then be used without burdening application writers with the concerns of field experts.

In some ways, it can be said that this aspect of our approach bears some similarity with Google's new Prediction API[16]. This Google service allows application writers to train and use classifiers on their own datasets without requiring any knowledge of machine learning or data mining. The main distinction to our proposed work, however, is that in our system we rely on the highly-structured nature of physical information to relieve the application writer from the responsibilities to even perform the training, classification, and interpretation steps. Instead, our semantic models allow the middleware to perform all of these transparently, in the background, without ever burdening the application with the internal details of this process.

A.2. Middleware Overview

We envision that the Internet of Things will intersect with the Internet of Services leading to services that are aware of their surrounding physical environment. To achieve this goal, we adopt a service-oriented architecture that allows our middleware to abstract sensors/actuators and provide their functionalities as services. With this, we support the needed interoperability and flexibility within the Internet of Things through a loose coupling of components and orchestration of services. We use a semantic approach to represent devices, data, and their physical attributes, in order to answer high-level queries. Sensors and actuators are represented in an device ontology, and a domain-specific ontology is used to link keywords to related devices. Note, however, that we initially focus on sensors only, leaving actuators for future work.

[16] http://code.google.com/apis/predict/

The overall architecture for our IoTS middleware is shown in Figure A-1. As can be seen in the figure, the core part of our middleware is the Composition & Estimation module. It receives sensing and actuation requests from applications, interprets them using a semantic Knowledge-Base, maps them to the existing network topology, optimizes the execution dataflow, executes it, and provides a result. Throughout this process, our main **research contributions** lie in the following characteristics of our system:

- **Probabilistic discovery**: *Discovery* is the ability to find the services that match a given set of desirable attributes (sensing modality, geographic location, error characteristics, etc.). However, more than that, to address the challenges of scale mentioned in the beginning of this appendix we introduce the concept of *probabilistic discovery* as a tool to find the set of services that best approximates the one that is being sought after. This is described in more detail in Section A.4.
- **Smart expansion:** Given a description of the input parameters and of the desired output, *composition* consists of finding a dataflow graph that connects the available services in order to produce the desired output from the parameters. Within the composition process, our key research contribution are the use of *smart expansion* and *approximately-optimal composition*, as described in Section A.5.
- **Automated estimation:** By applying physical/statistical models on the historical spatiotemporal dataset from a set of services, our middleware estimates the most likely true value of the data at a given spatiotemporal point. This is done in a fully-automated manner, allowing people who are not experts in signal processing or estimation theory to fully utilize the IoTS. This is the subject of Section A.6.

In addition, a contribution that is fundamental to all others listed above is a **comprehensive set of ontologies** describing sensors, actuators, physical concepts, physical units, etc., as well as spatiotemporal and statistical correlation models of the data. In Figure A-1 this set of ontologies is called by the name “Knowledge Base”, consisting of three parts: a domain ontology, an estimation ontology, and a device ontology. We describe these in greater detail in Section A.3 that follows and in Appendix B.

A.3. The Semantic Knowledge Base

As mentioned earlier, in our IoTS middleware three distinct ontologies comprise what we call collectively as a Knowledge Base. To give a high-level view of the structure of this Knowledge Base, we briefly outline its main classes below.

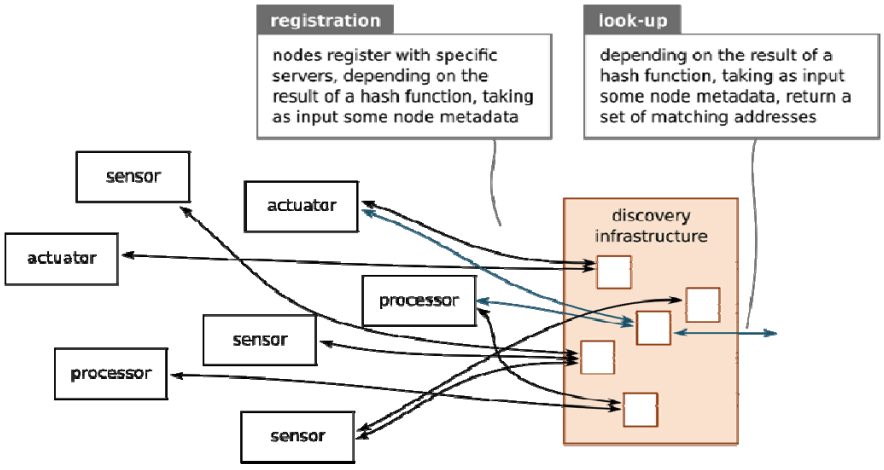

Figure A-3. The infrastructure related to the discovery process can be abstracted as a data-store that supports two instructions: insert (for service registration) and query (for service look-up).

- Device Ontology: The device ontology stores information regarding actual hardware

devices that may exist in the network, including manufacturers, models, type of device, etc., and connects each device to related concepts in other ontologies (for example, which physical units it uses when outputting data, etc.). The main subclasses of this ontology are:

1. Sensors
2. Actuators
3. Processors
4. Composite (devices)

- Domain ontology: The (physics and mathematics) domain ontology carries information about how different physical concepts are related to one another. For instance, in Figure A-2 the “wind chill” concept is shown to be a function two other concepts: “temperature” and “wind speed”. This ontology also links each physical concept with a set of physical units (“km/h”, “m/s”, …) as well as with the known sensors/actuators that can measure/change them. In Figure A-2 the physical concept of “location” is linked to the device ontology by declaring it to be measurable by GPS, by WiFi fingerprinting, and GSM triangulation.

  The main subclasses of the domain ontology are:

  1. Physical concepts
  2. Physical units
  3. Data structures
  4. Mathematical formulas

- Estimation Ontology: The estimation ontology contains information about different estimation models (“linear interpolation”, “Kalman filter”, “naïve Bayesian learning”, etc.), the equations that drive them, the services that implement them, their QoS, and so on. These are subdivided into:

  1. Data models
     1. Spatio-temporal
     2. Statistical
  2. Error/Uncertainty models

Note that each of the branches above contains several levels of subclasses, attributes, and instances, which are omitted here. One of the most important tasks that we have yet to complete in this project, is to clearly specify the sub-structure that we have omitted above. This is currently in progress.

Building upon these ontologies, the three core parts of our system (discovery, composition, and estimation) are described in the three sections that follow.

A.4. Discovery

Discovery Problem: Find the services in the network that match a given a set of desired attributes.

In networks of unknown topology, it is the job of a *discovery* layer to inform an application how to address each desired service. In this process, applications describe the service that they are looking for, and the discovery layer returns the set of devices that best match that request. At its core, the device discovery is composed of two parts:

- **Registration**: Registering services either as they come online or as they are discovered;

- **Look-up**: Querying the network to find desired services.

We contend that, in traditional discovery scenarios, much of the infrastructure related to registration and look-up can be abstracted as a general data store. This is shown in Figure A-3. That is, the act of registering a new service can be abstracted as inserting the service's address and metadata into the data-store, and the act of looking a service up can be abstracted as *querying* the data-store for the services that match a set of metadata attributes. Hence, the infrastructure for these two parts inherit much of the work that already exists in the Databases domain. To address the *scale* challenge, which can cause the data-store to grow too large and unwieldy, we propose to modify the traditional discovery process with the addition of non-deterministic behavior at several levels, as described next.

Probabilistic Discovery

The key difference between our discovery layer and those in the literature is the support for what we call probabilistic discovery, which is meant to allow our middleware to continue to operate within reasonable time, memory, processing, and power constraints as the network size grows into the millions. To better understand the concept of probabilistic discovery, let us return to the example from the introduction: an application requires to know the average air temperature at the city of Paris at this very moment, and it is up to our middleware to find a reasonable approximation. For this, the middleware fetches the definition of “average” from the ontology, which includes a description of the well-known equation for the sampling distribution of the mean. This equation states that in a network of M sensors, we can afford to instead use only N sensors ($N < M$) to calculate the average temperature within some mean error of e. Then, the job of the discovery layer is to pick a set of N temperature sensors in Paris that is uniformly-distributed in space in order to provide the result with error e. For this, the discovery layer must perform the following actions:

- Use the error equation to estimate the number N of sensors that will be needed for this request.
- Produce a random sample of N points in time, space, and other dimensions (such as sensor/actuator orientation in space, their coverage area, or any other attribute of a device).
- Discover the N devices in the network topology that best match those N points.
- Recalculate the error given this set of devices to produce the final error estimate.
- Possibly repeat 2-5, depending on whether the final error estimate is satisfactory.

The example above is an instance of what we call a **probabilistic lookup**. That is, an intelligently-constructed query that makes use of probabilities to look up approximate information when exact values would be too costly to compute. In a similar manner, another aspect of discovery that can also be made probabilistic is the registration process. In **probabilistic registration**, services use non-deterministic functions to determine (1) *at what times* registration should take place, (2) *which server* it should register with, and (3) *what attributes* it should register with each server.

Part of our upcoming research will consist of characterizing the different combinations of probability distributions that can be used throughout the discovery process (in both the probabilistic query and probabilistic registration), in order to establish what are the advantages and disadvantages of each discovery scheme.

A.5. Composition

Composition Problem: Given a description of the input parameters and of the desired output, find a dataflow graph that connects the available services in order to produce the desired output from the parameters.

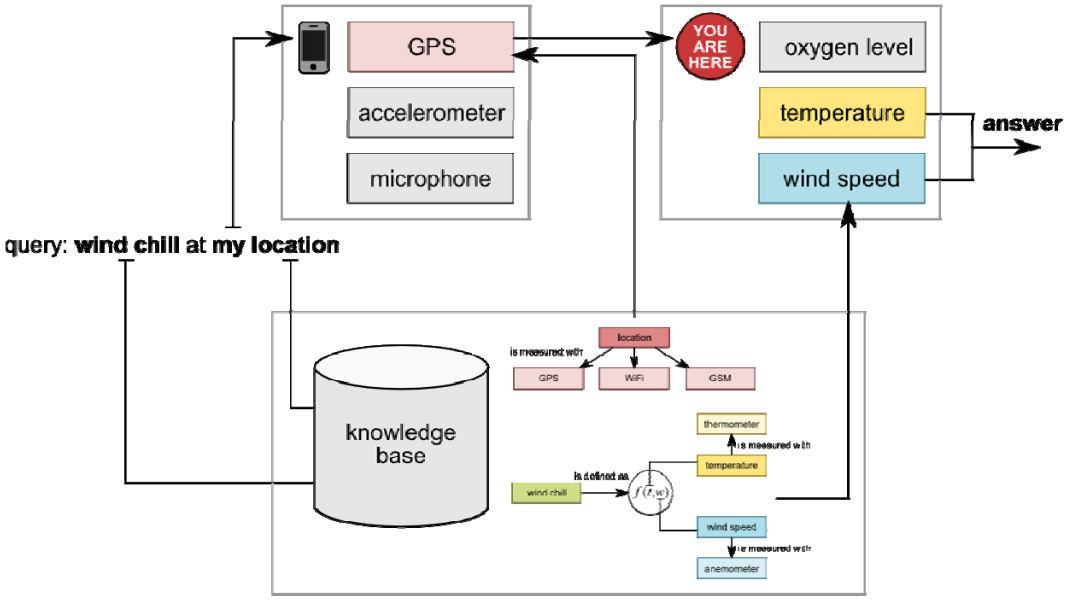

Figure A-4. Example of "composition" in the Internet of Things.

Considering the problem summary above, from now on we will use the word "query" to denote the input to the composition problem. That is, a *query* is a description of the input parameters and of the desired output. Thus, given a query, the middleware must compose services into a dataflow diagram originating at the described input and ending at the desired output.

Figure A-4 illustrates the main idea behind the composition process. In that example, a moving user wishes to keep track of the wind-chill factor at his location. He has no knowledge of the underlying sensors, actuators, or any technical information that is required to fulfill his request. Instead, he accesses an end-user application which takes a query from him and contacts our middleware in order to compose the required services and provide a solution. More specifically, it is up to the IoTS middleware to identify all services in the current network topology which can provide (in this example) *location* measurements, as well as all sources for *wind-chill* measurements. For wind-chill, it is found that by using a certain mathematical function it is possible to calculate the wind-chill based on temperature and wind-speed data, provided by a thermometer and an anemometer respectively (Figure A-2). For location, the middleware discovers that the user is carrying a location-capable device (a GPS), and uses it to pinpoint his/her latitude and longitude. With this, the middleware can search for wind-chill–compatible services in the network topology, thus discovering the existence of a thermometer and anemometer nearby. Finally, these two sensors are sampled, and a response is calculated by applying the wind-chill formula on the two resulting measurements. Part of this process, then, repeats every-so-often to continue providing the user with information about the wind-chill around him as he changes his position in the environment.

From the example, it is clear to see that composition queries can be of two types, regarding their duration: **one-time queries** are those that return data to the requester only once, similar to how standard database queries operate; and **continuous queries** are those where the results are returned at multiple different times, executing either forever or for a limited lifetime. The wind-chill example is an instance of a continuous query. Another example of a simple continuous query is "give me the value of the temperature sensor in room 1 every 2 minutes".

In addition, differently from most database systems, queries may also specify **sequences of conditions** (instead of just conjunctions and disjunctions). For instance, "if X=1 and 10 minutes later Y=2, do *action1*(); Then, if Z becomes < 3 within 5 minutes, do *action2*()". As such, composition queries differ from standard database queries in that they incorporate a

concept of time and sequential ordering. In the literature, spatio-temporal extensions of databases are handled by data-stream management systems (DSMS) such as [CCEG03, ABBC03 MVOP08, SPDM08, WXCJ98, KGB10], and complex-event processors (CEP).

An exact solution to the service composition problem can be achieved in a brute-force manner through the four steps listed below, and pictured in Figure A-1. Note that while this solution is mathematically correct, it clearly does not fit in the large-scale Internet that we envision, as we anticipate the ontologies to be too large, and the services too numerous. Instead, we plan on adopting an approximate approach, as detailed later. In the meantime, though, the brute-force, exact solution to the composition problem can be described as follows:

- **Expansion**: This step expands the initial query by replacing each term with an equivalent expression, found by traversing the domain ontology. Every term in the query is mapped to a concept, in the ontology, that in turn can be substituted with one or several sub-concepts. Sub-concepts can be either a (set of) mathematical function(s) that is equivalent to the concept itself or an information source that can provide real-world information about its parent. The query expansion process stops when leaf nodes in the ontology are reached, to provide, as a final result, a set of all possible combinations of service dataflows that answer the initial query.
- **Mapping**: This step takes all queries identified in the expansion step, above, and maps them to the actual network topology. As such, mapping is necessarily performed by interacting closely with the service discovery layer. When the set of queries is received, this step decomposes each query into atomic sub-queries that can be mapped, each, to a single service in the network. The query takes advantage of our device ontology, that models real world devices, to complement any missing information during the discovery process. If a query cannot be mapped, due to a service not being present in the network, or the requester lacking the necessary permissions, the system reacts by dropping that query from the expanded set. The output of the mapping step is the set of all possible dataflows that can answer the expanded queries (and, therefore, also the initial query) in the current network topology.
- **Optimal mapping selection**: Once all possible dataflows have been defined (mapping step), the IoTS middleware must choose one dataflow to enact. In this step, therefore,we find one dataflow that is, in some predefined way, optimal, and pass it to the execution block, below.
- **Execution**: Now that the best composition of services has been determined for the query, in the execution step the services are actually accessed and the result is returned (or stored). In addition, during execution, the middleware must check for any conflicts that may arise at run-time, in a process that we call multi-agent conflict resolution. This is currently left as future work, when we will develop access control policies to manage concurrent access to service providers, especially actuators.

Approximate Composition

As mentioned earlier, the brute force approach to composition is not suitable for the large scale of the Internet of Things. As an alternative, therefore, we propose an approach that avoids calculating all equivalent sets of queries (which grows very large very quickly), by modifying only the expansion and mapping processes from the brute-force approach:

- **Smart query expansion**: To avoid exhaustively calculating all possible equivalent sets of data-flows, only one of which will eventually be selected during the optimization phase, a much smarter approach to expansion is to instead produce a reduced set of good candidates. To perform this step, we assume certain knowledge of the network including the estimation of the category or number (or other attributes) of available services. Based on these estimations, a computation function will produce a set that

contains the candidate queries along with the time needed to execute each query and the number of services it requires. These candidates are the dataflows that have the highest likelihood of having a matching overlay in the network that satisfies a set of predefined constraints. An example constraint could be that the predicted execution time should fall within a certain acceptable interval.

- **Probabilistic mapping**: Taking as input the set of candidate dataflows from the previous phase, the probabilistic mapping phase differs from regular mapping in that it does not attempt to find all possible mappings of the input dataflows into the network topology. Instead, this phase will randomly pick a small subset of all implementable mappings by making small, atomic queries to the probabilistic Discovery module. The result is a much reduced set of dataflow mappings that are computed in considerably less time and using (hopefully) orders of magnitudes less resources. Similarly to the brute force based query mapping, probabilistic mapping will utilize the device ontology to complement the network's knowledge of services in order to facilitate the composition process.

Data Streaming and Storage Support

Once all the steps above successfully take place, the system still has to execute the original request by enacting the optimal dataflow found in the previous step, and returning the results as appropriate. In this process, it will often happen that in addition to the latest sensor measurements, some amount of past data will also need to be accessed.

Therefore, it is of utmost importance that the IoTS middleware supports not only live data streams but also historical data where measurements and other information are stored for later access. Although we have not yet started our work in this area, the literature includes several solutions that lay the groundwork for the main requirements that this imposes on our middleware: data storage, continuous queries, temporal queries, and spatial queries. One example is TinyDB [MFHH05], which support in-network processing and sensor queries. The advantage of TinyDB is that it is deployed and widely used although it supports only limited aggregate functions and does not support injection. It also does not focus on scalability, nor mobility and it does not provide long-term storage. Data Centric Storage [RKSE03] is another implemented solution that stores data on the nodes closest to the event's location and stores replicas on the surrounding nodes but it gets inefficient when the number of events increases. TSAR/PRESTO [Desnoyer05] supports in-network storage of data but requests information from sensors only if their readings differ from the reading prediction model built by proxies, which are powerful nodes in the network that are in charge of one cluster of sensors each. TinyPEDS [GWMA06] also supports in-network storage where data is stored on elected cluster-heads, which store the data of the cluster they are responsible for.

A.6. Estimation

Estimation Problem: Using physical/statistical models and historical spatiotemporal dataset from a set of devices, estimate the most likely *true value* of the data at a given spatiotemporal point.

Except for the simplest situations, estimation is currently a process that is performed in a rather manual manner. As it is, it befalls a system designer to search the literature for a fitting model to use, extract deployment information, calibrate data sources, train classifiers, fill in matrices of parameters, etc. Then, when another system designer faces a similar situation, he is again tasked with repeating most of that work for his own system. At best, this constitutes a clear duplication of labor. But more than this, as new devices are added to the system, as old ones are removed, and as existing ones slowly deteriorate, much of this laborious process must be repeated every-so-often, to keep the system's error margins in check. Further, in a system of unknown topology, taking a full-knowledge design approach in

this manner is simply not possible.

As a solution, in our IoTS middleware, we propose to automate this entire process. For this, we once again rely on a large ontology of device types and physical concepts. However, more than that, for this purpose the ontology must be augmented with error models, spatiotemporal propagation models, and statistical correlation models. With all of this information, solving the estimation problem becomes a 3-step process:

- **Model discovery**: Searching the ontology for all the models related to the devices providing the desired services.
- **Optimal model selection**: Picking the most appropriate models based on a few parameters and a cost function (also specified in the ontology).
- **Estimation execution**: Applying the models to the existing historical data from sensors, using as input parameters the sensor and deployment metadata. This will be done using pre-developed engines for each model.

In addition, to make sure the metadata is up-to-date, the middleware can also **continuously execute recalibration procedures** on the sensors by applying calibration models described in the ontology to historical sensor data. In many cases, even, the same spatiotemporal and statistical models used during estimation can also be employed for calibration. This is because the estimation process is a function *f: parameters* × *input data* → *estimated values*, while calibration is a function *g: true values* × *input data* → *parameters*. That is, the two processes are duals of one another, so the entire estimation process can be made self-correcting.

Note that the key to the estimation procedure described above is that we do not propose a one-size-fits-all solution to the estimation problem. Instead, we heavily rely on models that are entered by field experts into the device ontology. These same experts also develop processing engines, such as a Kalman filter engine, a particle filter engine, a naïve-Bayes classifier engine, etc., which act as plug-ins to the IoTS middleware in order to enact the models. This will make the middleware agnostic to the different estimation solutions and, in a major way, future-proof.

A.7. Current status: Mobile-phone-level middleware

To understand the fundamental requirements related to modeling the different types of devices in a network and creating the related ontologies, we initially narrowed down our target from the domain of large sensor/actuator networks to that of sensors residing within the same physical device. That is, our initial focus has been on modeling the different sensors in current mobile phones.

This bottom-up approach is interesting for two reasons: first, it helps us solve the modeling problem at its simplest form, without the distractions of specifying communications protocols and so on. But, secondly, this approach also lays the groundwork for the deployment of large-scale sensor/actuator networks, by using the largest class of sensor/actuator-carrying devices on the Internet at this moment: mobile phones.

Our solution is written in Java, with initial target OS being Android. Architecturally, the code is organized as shown in Figure A-5, that is:

Applications access sensors in the system by instantiating a **Thing/Sensor Mediator** object, which is in charge of accessing a **Thing/Sensor Daemon** singleton running in the system. Through the Mediator, the application may perform basic discovery and sensing actions, including: discovering all sensors in the device, discovering sensors that belong to a certain class in the sensor ontology, sampling a desired sensor, setting up periodic sensing tasks, and setting up event-based sensing tasks. Each of these is done with a single method call, abstracting away all the logical details such as creating separate processes in the OS, executing inter-process communication calls, synchronization, etc. All of these lower-level

issues are all handled by the Mediator and the Daemon in an OS-independent manner. In addition, all sensors are treated homogeneously under a common API, whether they are microphones, thermometers, accelerometers, or cameras.

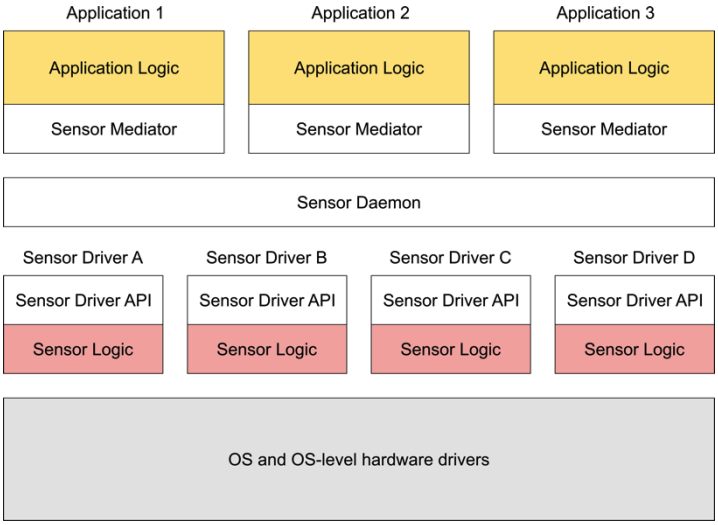

Figure A-5. Mobile-phone middleware architecture.

For this, at the other end of the Daemon's responsibilities is the task of communicating with the sensors themselves. This is done through the use of plug-ins called *Drivers*.

A **Thing/Sensor Driver** is an OS-specific piece of software that abides by our Sensor API by implementing the Sensor interface. This interface defines a small set of low-level methods for extracting sensor metadata and sampling a sensor. Each Driver declares some simple metadata, such as full name, version, author, etc., as well as two properties that tie the driver into our device ontology. These properties are the sensor ID (for instance “com.mycompany.cameras.Supercam123”) and an interface (“org.standardsbody.sensors.cameras.ConsumerCameraStandardV2”).

The sensor ID is a key with which the exact entry for this specific Sensor Driver's maker and model can be found in the ontology. Similarly, the interface ID points to the exact entry in the ontology where a specification of the sensor's data structures, data ranges, physical units, etc., can be found. Therefore, any two sensors that implement the same interface are, for most intents and purposes, interchangeable.

All sensor data is transferred using a common **Sensor Data** object, which can be extended and customized for each sensor interface. The common information carried Sensor Data instances are the ID of the sensor that produced it, and the timestamp of when that specific data point was sampled. Sensor Data objects also carry their own tie-ins to the device ontology, through the form of a datatype ID, similar in purpose to the sensor ID.

Data flows from the sensor hardware, through the OS-specific handlers, to the Sensor Driver, the Sensor Daemon, the Mediator, and, finally, reaches the application that requested it. When this happens, data is also attached a status code to inform the application of any errors or warnings that may have been triggered in the process.

All of this is already implemented. The next step in the development of this mobile-phone middleware is to add support for data *sinks* (while sensors can be seen as data *sources*) which allow us to extend the scope of our mobile-phone middleware into applications that require actuation and communication with other services through the Internet.

A.8. Conclusion

We have presented our vision for a middleware for the Internet of Things-based services. Our middleware allows applications to request information from heterogeneous sensor/actuator service providers distributed on a global scale. We adopt a service-oriented

architecture to abstract all sensors and actuators as services in order to hide their heterogeneity. Our approach is heavily based on a Knowledge Base that carries information about sensors, actuators, manufacturers, physical concepts, physical units, data models, and error models. To address challenges of scale and heterogeneity, we concentrate on three core research contributions: probabilistic discovery, functional composition, and estimation. Together, these three contributions allow our middleware to respond to sensing or actuation requests while managing the complex relationship between accuracy and time, memory, processing, and energy constraints.

Appendix B – Ontologies for IoTS

In this appendix, we focus on modelling a set of ontologies that describe devices and their functionalities and thoroughly model the domain of physics. The physics domain is indeed at the core of the IoTS, as it allows the approximation and estimation of functionalities usually provided by *things*. Those functionalities will be deployed as services on appropriate devices through the middleware presented in Appendix A.

Following, in Section B.1 we discuss related work and point out our contribution with respect to these efforts, in Section B.2 we detail the proposed ontologies for IoTS and finally in Section B.3 we conclude our discussion on the proposed ontologies.

B.1. Related Work

Semantic technologies within the IoT context are perceived as having three main benefits: high-level abstractions of complex information and incremented knowledge that provides a support for service composition and better interoperability.

Abstractions in IoT solutions, that integrate ontologies in their approach, can be divided in two categories: On the one hand, some IoT projects take advantage of ontologies as they abstract devices as services (such as in HYDRA[ZH09, ERA10], SENSEI [PBEV09]). On the other hand, others use them as they abstract data/information as services (among which are SOFIA [HLBT10], SATware [MDMV09], Global Sensor Networks GSN[AHS07], and Sensor-Masher[PH09]). A common approach towards this purpose is the use of virtual/semantic sensors [ZH09, ERA10, AHS07] to abstract one or several physical devices. Similarly, [WZL06, PH09, MDMV09] adopt the concept of semantic sensors and semantic streams. However, their implementation is different as they focus on abstracting data streams into higher level semantically rich knowledge. Semantic devices provide composition in some manner where the composed functions are specified at design time, and the mapping onto the network devices happens dynamically at run time.

To provide better interoperability, three aspects of the real world are modelled thoroughly in ontologies created within IoT solutions: *things* [PBEV09, ERA10], information and reasoning over data generated by *things* [ELS07, ERA10, Phoc09], and services [ZH09, ERA10]. Some projects go a step further by using ontologies to model context information [PBEV09], or dynamic reconfiguration, and adaptive resource management [KKKN08]. The target in [PH09], however, is the integration of sensor data streams into the World Wide Web rather than into an Internet of Things.

It should be noted that none of these solutions try to model and combine knowledge domains representing the real world into one global ontology as we do, to address the challenges presented at the beginning of this Appendix. They are however, mostly focused on modelling their ontologies for specific purposes only. Further, it is not clear how any of those ontologies are modelled to address scalability.

B.2. IoTS Ontologies

An ontology is defined as “a formal, explicit specification of a shared conceptualization” [GCG94] and is used to represent knowledge within a domain as a set of concepts related to each other. There are four main components that compose an ontology: Classes, relations, attributes and individuals. Classes are the main concepts to describe. Each class can have one or several children, known as subclasses, used to define more specific concepts. Classes and subclasses have attributes that represent their properties and characteristics. Individuals are instances of classes or their properties. Finally, relations are the edges that connect all the presented components.

Global Ontology for The IoTS

We envision the representation of the IoTS-based real world to be divided into 3 layers: a physical layer, i.e., *things*; an information layer, i.e., data and metadata about knowledge provided by *things*; and a functional layer comprising services provided by *things*. To match our vision of the real world and its representation by the Internet of Things, we aim at building an ontology that actually models all three layers. In fact, the physical layer is represented by the **Device Ontology**. The information and service layers are represented by the **(Physics and Mathematics) Domain Ontology** and **Estimation Models Ontology**. To describe the ontologies more precisely:

Device Ontology: The **Device Ontology** models actual hardware devices that may exist in the network. For our middleware, it can be regarded as the device description repository that can be accessed for discovery.

Domain Ontology: The (**Physics and Mathematics**) **Domain Ontology** models information about real world physical concepts and their relations among each other. For our middleware, it can be regarded as the main repository to access for service composition.

Estimation Ontology: The **Estimation Ontology** contains information about different estimation models ("linear interpolation", "Kalman filter", "naive Bayesian learning", etc.), the equations that drive them, the services that implement them, and so on. For our middleware, it can be mainly regarded as the repository describing the device's quality of service, and provides information needed for service composition. We aim at providing this ontology to be used as a reference by any middleware or application requiring IoTS services, i.e., services provided by real world *things*. Those services, in most cases, generate approximate but never 100% accurate outcomes.

Most existing ontology work focused on modelling either devices as done, e.g., in MMI[17] ontology and [NC09, Gomez08], or physics [Kuhn09, CMNG08] separately. The novelty of our approach is that it combines and takes advantages of the three ontologies by linking, all together, the domain of knowledge for sensing, actuating, and processing tasks and the real world representation through IoTS services, that are aware of their environment. An important contribution is the level of abstraction at which we represent *things*, as we allow users to describe devices in an expressive manner while still avoiding complex details. In fact, as we target scalability, we consider simplicity in modelling knowledge to be an essential criteria. We argue that too much details might hinder the readability and quick traversability of the ontologies, thus effecting their scalability and usability. Of course the full ontologies are too large to be described in this appendix. So, in the following, we outline only the most important concepts.

Device Ontology

As mentioned earlier, the Device Ontology is accessed to identify what *things* should be looked up to satisfy an application's requirements. We consider that applications built on top of an IoTS middleware should be network and device agnostic. Therefore, it becomes the task of the middleware to identify what devices to seek in order to provide needed services. For this purpose, the ontology should clearly describe and yet not over-specify device metadata.

Our main contribution is the **high-level abstraction for device metadata**, especially regarding the internal components of devices. Internal components are the electronic chips and hardware parts, built inside the device, that together define its technical functionalities. Hence, looking at each of them separately as independent entities is not informative, as their functionalities are tightly related to one another's. That being said, understanding the

[17] http://marinemetadata.org/community/ teams/ontdevices

characteristics of a singular chip requires an understanding of the whole device's internal schema, which grows to be too complex to include. We can further argue that they can just be considered as a black box, especially that those components are not directly accessible by applications. However, we chose to allow users to describe the internal components of devices (also done in [NC09, Gomez08]) by their name and type only.

Another main contribution is that our ontology **holds knowledge that is independent of device deployments**, e.g., information related to the device's actual location. Instead, deployment information is presented in the metadata, reported by devices, during the discovery registration process. The ontology becomes thus easily pluggable with any middleware or application.

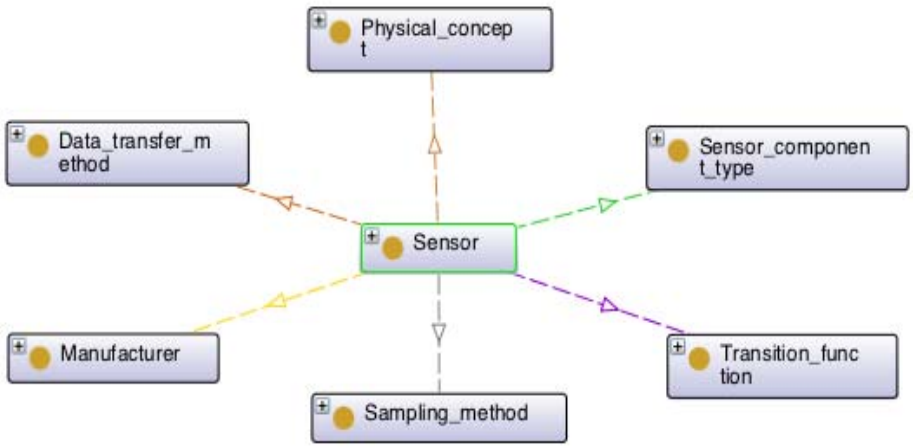

Figure B-1. Sensor and related first class entities.

To elaborate on the ontology, we consider that IoTS devices can be divided in four main classes:

- *Sensor*: A device that has the capability to measure a physical property of the real world.
- *Actuator*: A device that has the capability to perform an operation on or control a system/physical entity in the real world.
- *Processor*: A device that has the capability to perform computation operations on data.
- *Composite*: A device that consists of at least 2 of the devices above.

In the following, we focus on modelling sensors, as they are representative of *things* and models of other devices adhere to the same conceptualization approach. Based on the current literature [ELS07, LZ05], we have identified several ontology concepts that are commonly used to model sensors (sensorML[18]). As shown in Figure B-1, those concepts are:

- *Manufacturer*: The manufacturer of the sensor.
- Sensor component type: The sensor internal hardware components.
- *Physical concept*: The real world property measured by the sensor (e.g., temperature, wind speed, etc.). This concept is the main link between the Device Ontology and the Physics Ontology.
- *Sampling method: The way the sensor is triggered to sample its environment (e.g., periodic).*

[18] http://www.opengeospatial.org/standards/sensorml

- *Data transfer method: The way the sensor is triggered to transfer its readings (e.g., push).*
- *Transition function: The process used to convert the input phenomenon to a digital value.*

We chose the entities above as we consider that a sensor can be properly identified given any of their respective values. With the exception of the last entity, which we introduce as it clarifies what and how phenomena or values are output by a sensor after a measurement is performed. This is needed so that a sensor's outputs can be meaningful to and usable by other applications.

Physics Domain Ontology

The **Physics Domain Ontology** is created with two main goals. The first is to model real world entities as physical concepts so that any IoTS middleware can extract knowledge about the real world, as this is a common task to be performed within the Internet of Things. The second is to model mathematical formulas and functions as they are the first alternative to be utilized when no device can provide needed services. The main classes of the Domain Ontology are:

- *Physical concept: A real world object or property that can be measured.*
- *Physical unit: The output unit of the real world property measurement.*
- *Mathematical datatype: The set of numbers that can represent a real world property measurement.*
- *Formula: Mathematical expression that computes a numerical value representing a real world property.*
- *Function: Formulas are implemented by functions that define the required input and output machine datatypes.*

Our main contribution in this ontology is that **we model and establish a direct relation between physical concepts, mathematical formulas and functions**. We argue that this relation is essential as it allows services to be provided as mathematical computations over physical concepts. This process can be used by any middleware to substitute services of unavailable devices with alternative services that can be deployed on any other appropriate device, which is a very familiar scenario within the highly dynamic IoTS. This relation further allows our ontology to be useful in any context requiring mathematical and physical knowledge by clearly modelling formulas that can compute mathematical values as measurements over a physical property.

However, a same physical concept can have several formulas that vary based on the units of measurement of input/output parameters. Hence, our second contribution is to introduce two, not previously described, first class entities: unit constraints and conversion formulas. The former allows users to specify if a formula can have only one output and one input unit per concept, or can have a defined set of such units, or it stands correct for any input/output units, linked to a physical concept. For instance, the formula speed = distance/time stands correct for any distance unit over any time unit. On the other hand, a windchill formula for temperature in Celsius and wind speed in km/h is different than that of a temperature in Fahrenheit and wind speed in mph. As for the conversion formulas class, it allows users to model conversion formulas between one measurement unit to another. By adding the constraints class and conversion functions to our ontology, we introduce a higher degree of flexibility as it allows any middleware or application using our model to dynamically adapt to unit constraints.

A common reference model for representing and categorizing physical concepts is the

DOLCE[19] representation. It is adopted by several works such as [DNJC10, Kuhn09], as it has a well-organized vocabulary. However, it does not categorize entities by their physical properties but rather by the human perception of those entities. This organization is not in line with our representation of concepts that should be both intuitive and physics oriented. In our ontology, each physical concept is linked to:

- *Sensor*: All sensors that can measure its value.
- *Units of measurement*: All units by which it can be measured.
- *Mathematical datatype*: Set the mathematical values the concept can be represented
- *Formula*: All mathematical formulas that can compute its value.

Regarding formulas, authors in [CMNG08] provide an approach to modelling physics in an ontology. However, their model is only applicable for the biological domain as they focus on mapping laws of physics to biological processes. SPACE is another ontology that models physics but it only applies in the space physics domains [NSM10]. In our ontology, each formula is linked to:

- *Mathematical expression: Mathematical equation.*
- *Input parameters: Measurements of physical concepts that will be used to evaluate another physical concept and their measurement units.*
- *Output parameter: Computed output and its measurement unit.*
- *Physical concept: The physical concept being evaluated.*
- *Unit of measurement: It is in fact the output's unit.*

We argue that those concepts are well representative of physics and mathematical models and they specify all the parameters needed to define a mathematical equation. The Sensei project [PBEV09] models a decomposition of physical concepts into a set of other physical concepts. This decomposition can be similar to a direct link between our formula output and input concepts. However, the relation between their concepts is not clearly specified and therefore, their decomposition cannot substitute our formula model.

Estimation Ontology

The **Estimation Ontology** is, perhaps, the most unusual among the three described here. This ontology is in charge of storing the different mathematical models that make up the mental toolbox carried by expert system designers in fields such as Robotics, Estimation, Sensor Networking, etc. However, in addition to simply storing these models, the Estimation Ontology must also organize them in a way that makes them machine-accessible. After all, our middleware must be able to discern (1) which models are appropriate for a given situation, and (2) which model is, in some sense, optimal.

For this reason, the Estimation Ontology must provide a well-designed set of attributes for each model, as well as an intricate web of relationships between models, devices, and physical concepts. These rather indispensable elements, and how to best express them in this ontology, are something that we are currently investigating. For now, we limit ourselves to providing below a first-level set of classes that group the different types of mathematical models:

- *Estimation & Prediction Models: These models are used during the automated estimation process, as well as before the look-up phase in probabilistic discovery. An example of such a model is a Kalman filter where each component in its matrices and vectors are functions of physical concepts from the Domain Ontology (for instance, an*

[19] http://www.loa-cnr.it/DOLCE.html

input vector for use in target-localization could be defined as a triplet of 3D-acceleration, 3D-velocity, and 3D-position).

- *Association & Correlation Models: These describe the numerous conditional probability relations that are used in Estimation Theory, relating one physical phenomenon as a function of another. For instance, the probability of the value of a temperature sensor given the value of a daylight sensor. In addition, these models can also be used to solve the Association Problem that occurs, for instance, with multiple-target tracking.*
- *Error Models: These models describe the different ways that uncertainties can be introduced into measurements and actuations. These are usually represented in the form of a stochastic model (for instance, a simple additive Gaussian noise model).*

B.3. Conclusion

We presented in this appendix a Global Ontology we are building for IoTS. The ontology models three aspects of the real world present in the Internet of Things. The first aspect is the "*things*" aspect described in a Device Ontology. The second aspect consists of real world concepts and functionalities of things, modelled in a Domain Ontology as mathematical formulas, and third is a real world approximation aspect that describes models to be used to approximate unavailable services and estimate missing information. The proposed ontology is at the core of a Service Oriented middleware for the Internet of Things, we are developing, that is scalable, flexible and provides the needed interoperability between deeply heterogeneous IoTS components as detailed in [THVG11].

Our future work will consist of further investigating the sensor modelling approach on different levels of details, as we plan on performing deeper comparison with existing solutions. We pay special attention to SensorML as it provides an appropriate modelling approach, although too detailed for our purposes. We later plan on investigating actuator and processor modelling approaches, but we consider they will strongly adhere to sensor models. Furthermore, we plan on modelling the estimation ontology comprising spatiotemporal and statistical correlation models of data. As for the middleware solution, we plan on implementing our vision and integrating the ontology to evaluate its feasibility.

Appendix C – SCA technical information

Both the business and Thing-based facets of the SCA-based XSC rely on an SCA model. In this appendix we give definitions for the elementary concepts of this model.

SCA stands for Service Component Architecture, which comprises a set of specifications[20], actually stabilized at v1.00, aiming at providing the basis to create and compose services. These specifications have been written by a group of industrials called the OSOA collaboration[21] which counts among its participants the following companies: BEA, IBM, IONA, Oracle, Red Hat, SAP, Sun and Sybase.

The SCA specifications define most notably a language-neutral programming model and an assembly mechanism for components implemented using various technologies. The specifications also cover the non-functional requirements through the SCA policy framework.

The SCA binding mechanism allows to plug existing EJB sessions beans or JCA connectors into an SCA component, thus easing the integration of this technology in the enterprise world.

Philosophy and main concepts

In a SOA perspective, as in CHOReOS, building a business application consists of creating services matching a business need or using existing ones. SCA provides a programming model to encompass both these two aspects, the creation and composition of services.

The main idea of SCA is to see a business application as a set of components linked together. A component is actually a key concept in SCA: it offers *services* through an interface and consumes services from other components, called *references*. SCA also defines larger structures, known as *composites***.** Composites are assemblies of components, services, references, *properties* and the *wiring* between the different artefacts. These artefacts are all detailed in Figure C-1 that gives the anatomy of an SCA composite. They also are more formally defined in the SCA meta-model defined at OSOA.org (the official site of the SCA and SDO specifications) [OSOA07], as represented in Figure C-2.

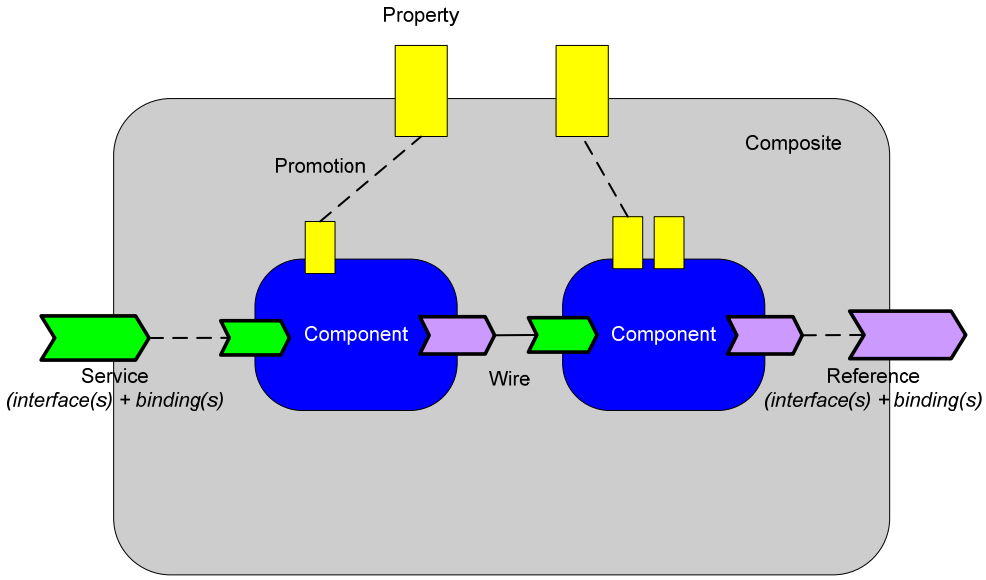

Figure C-1. Anatomy of an SCA composite.

[20] http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications
[21] http://www.osoa.org/

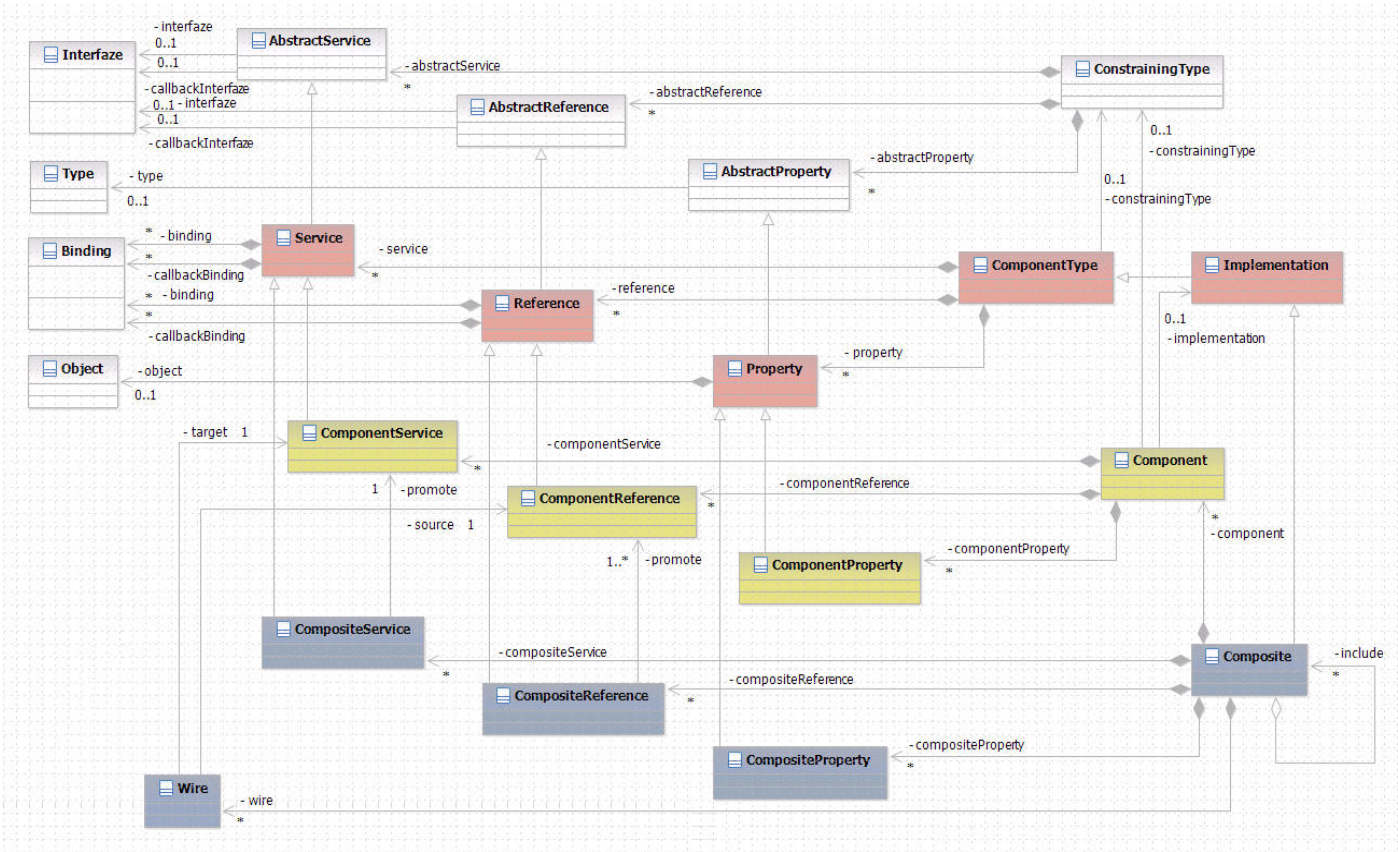

Figure C-2. SCA meta-model.

SCA embraces a wide range of technologies. Regarding to service creation, specifications exist to support the development of service components in different programming languages including Java, C++, Cobol, C and BPEL.

Likewise, SCA specifications also deal with the communication between services and provide *bindings* to do so. A *binding* describes the access mechanism to be called by a client for a service, and the access mechanism used to call a service for a reference. SCA currently supports the following binding types: Web services, JMS, EJB Session Bean and JCA. New binding types can be added through an extensibility mechanism.

The complete runtime configuration which contains components and composites is called the SCA *domain*. An SCA *domain* defines the boundaries for every component and composite. The boundaries are limited to a single SCA domain and cannot cross over multiple domains. An SCA component is therefore configured for a specific SCA runtime.

Communication between different domains and other external entities (such as Web services or Things) is still made possible through bindings. As such, SCA composites and components can interact with Web-based services using the existing Web services binding, which is SOAP based. Nevertheless a REST-based binding can also be implemented and tailored to CHOReOS needs.

In CHOReOS we further plan to specify and develop a binding that targets Thing-based services. This binding shall rely on a Device Profile for Web Services (DPWS).

SCA artefacts definition

In order to have a portable representation of its artefacts, SCA defines an XML file format which is used for the SCA domain configuration when deploying composites. This format is

referenced as the *Service Component Definition Language* (SCDL). For example, the following excerpt defines a composite with two components:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
      targetNamespace="http://example"
      name="ExampleComposite">

    <component name="Component1">
        ...
    </component>

    <component name="Component2">
        ...
    </component>

</composite>
```

Focus on Components and Composites

A *component* (Figure C-1) is a fundamental element to build a business application. It is defined by specifying:

- The component *implementation* (a component may have no implementation in a top down approach).
- The *services* it provides.
- The *references* it needs from others (components and composites within the same SCA domain and *external applications or systems*).
- The *properties* that configure data values in its implementation.

All these artefacts are declared in the SCDL file, as the following snippet shows:

```
    <component name="CalculatorServiceComponent">
        <implementation.java class="calculator.CalculatorServiceImpl"/>
        <service name="CalculatorService">
            ...
        </service>
        <reference name="addService">
            ...
        </reference>
        <reference name="subtractService">
            ...
        </reference>
        <reference name="multiplyService">
            ...
        </reference>
        <reference name="divideService">
            ...
        </reference>
```

```
        <property name="pi">3.14</property>
    </component>
```

The component implementation represents how a concrete *business* function is achieved. SCA allows implementations in a wide range of technologies. Currently specifications exist for Java, Spring, BPEL, C, C++ and Cobol.

The configurable part of a component is called the *component type*. It includes the services offered by the component, the references to other services and the settable properties. All these artefacts are made available by introspection of the implementation, as provided by the FraSCAti platform (see Section 5.1.2).

A *composite* (Figure C-1) is a SCA structure used to assemble components, services, references, properties and the wiring between these artefacts. A composite can also include other composites to build higher level business functions.

Composite services, as well as composite references are actually provided by components contained in the composite. This mechanism is called *promotion*. Within a composite, the components services and references can also be wired.

Focus on services and references

A service allows to represent what a component may provide to the others whereas a reference allows to represent what a component needs from the others, where the term others refers to components and composites within the same SCA domain and external applications or systems.

Defining a service or a reference consists in specifying both:

- *Interfaces,* to give a description of the business functions.
- *Bindings,* to access the business functions

An interface describes the business functions provided by a service and used by a reference. SCA currently supports the following interface types: Java/C++ interface, WSDL 1.1 PortTypes and WSDL 2.0 Interfaces. New interface types can be added through an extensibility mechanism.

A binding describes the access mechanism to be called by a client for a service and the access mechanism used to call a service for a reference. As mentioned before, SCA supports Web services binding types, among other binding types, and is also extensible concerning this aspect via a corresponding extensibility mechanism.

The FraSCAti SCA platform

SCA implementations are made available by four different open source communities:

- Apache Tuscany[22]
- Frabric3[23]
- FraSCAti[24]
- Service Conduit[25]

For the SCA-based XSC though, we rely on the FraSCAti SCA platform as it offers specific means for runtime manageability (including *introspection* and *reconfiguration*) of SOA applications and of their supporting environment [SMFD09]. While these features are not part

[22] http://tuscany.apache.org/
[23] http://www.fabric3.org/
[24] https://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti
[25] http://www.service-conduit.org/

of the baseline SCA specification, they are leveraged in our implementation to deal with functional and non-functional concerns.

Appendix D – CHOReOS Cloud & Grid API

In this appendix we provide further details concerning the API that is offered by the CHOReOS Cloud & Grid middleware. In particular, D.1 refers to the Node Pool Manager API, D.2 refers to the Service Deployer API and finally D.3 provides the Grid API.

D.1. Node Pool Manager API

Create new node

HTTP Method	URI	Request Content Sample	Responses
POST	/nodes	<node> <cpus>2</cpus> <ram>1024</ram> <storage>100</storage> <so>Ubuntu 10.4</so> <zone>eu-west-1</zone> </node>	201 CREATED <node link="/nodes/{id}"/> 500 ERROR

Delete an existing node

HTTP Method	URI	Request Content Sample	Responses
DELETE	/nodes/{id}	-	200 OK 404 NOT FOUND 500 ERROR

Modify an existing node

HTTP Method	URI	Request Content Sample	Responses
PUT	/nodes/{id}	<node> <storage>2</storage> </node>	200 OK 404 NOT FOUND 500 ERROR

Get existing node details

HTTP Method	URI	Request Content Sample	Responses
GET	/nodes/{id}	-	200 OK <node> <cpus>2</cpus> <ram>1024</ram> <storage>100</storage> <so>Ubuntu 10.4</so> <zone>eu-west-1</zone> <ip>20.10.9.8</ip> <hostname>h1.choreos.eu</hostname> </node> 404 NOT FOUND 500 ERROR

List existing nodes

HTTP Method	URI	Request Content	Responses
GET	/nodes?{search+criteria}	-	200 OK <nodes> <node> <cpus>2</cpus> <ram>1024</ram> … </node> <node> … </node> </nodes> 500 ERROR

The seach criteria can be anything like this:

{field1}={value1}&{field2}={value2}&...&{fieldN}={valueN}

Where fields are all available attributes in nodes (see Nodes attributes bellow)

Deploy a configuration to an existing node

HTTP Method	URI	Request Content Sample	Responses
POST	/nodes/{id}/configs	<config> <name>MYSQL</name> </config>	201 CREATED <config link="/nodes/{node_id}/configs/{config_id}"/> 404 NOT FOUND 500 ERROR

Get the configurations list from an existing node

HTTP Method	URI	Request Content Sample	Responses
GET	/nodes/{id}/configs	-	200 OK <configs> <config> <name>MYSQL</name> </config> <config> <name>APACHE</name> </config> ... </configs> 500 ERROR

Delete a configuration from a node

HTTP Method	URI	Request Content Sample	Responses
GET	/nodes/{id}/configs/{config_id}	-	200 OK 500 ERROR

D.2. Service Deployer API

Deploy a service

HTTP Method	URI	Request Content Sample	Responses
POST	/services	<service type="BPEL"> <codeLocation> URI </codeLocation> <resourcesImpact> <memoryImpact> light </memoryImpact> <cpuImpact> medium </cpuImpact> <ioImpact> heavy </ioImpact> <region> France </region> </resourcesImpact> </service>	201 CREATED <service link="/services/{id}"/> 500 ERROR

List deployed services

HTTP Method	URI	Request Content Sample	Responses
GET	/services	-	200 OK 500 ERROR

Modify service performance

HTTP Method	URI	Request Content Sample	Responses
PUT	/services/{id}	<service> <increase factor="200%"/> </service>	200 OK 404 NOT FOUND 500 ERROR

Undeploy a service

HTTP Method	URI	Request Content Sample	Responses
DELETE	/services/{id}	-	200 OK 404 NOT FOUND 500 ERROR

D.3. Grid Computing API

The Grid computing API will be used to access both the InteGrade middleware developed at USP and the Hadoop system developed by the Apache Foundation.

Submitting an application

HTTP Method	URI	Request Content Sample	Responses
POST	/applications/	<application> <name>VideoTranscoder</name> <description> Beta version </description> <architecture> Linux_i686 </architecture> *(application binary attached)* <hash>7cc…42</hash> </application>	201 CREATED <application link="/application/{id}"/> 500 ERROR

Modify an existing application

HTTP	URI	Request Content Sample	Responses

Method			
PUT	/application/{id}	<application> <description> Transcode mobile phone videos into other formats. </description> </application>	200 OK 500 ERROR

Delete an existing application

HTTP Method	URI	Request Content Sample	Responses
DELETE	/application/{id}	-	200 OK 500 ERROR

Get existing application details

This method works almost like returning what was sent to create the application, except that the application binary is not uploaded back to the user.

HTTP Method	URI	Request Content	Responses
GET	/application/{id}	-	200 OK <application> <name>VideoTranscoder</name> <description> Transcode mobile phone videos into other formats. </description> <architecture> Linux_i686 </architecture> <hash>7cc…42</hash> </application> 404 NOT FOUND 500 ERROR

List existing applications

This operation lists applications related to the term specified by the “query” value. If “query” is empty or missing, all applications are listed.

HTTP Method	URI	Request Content	Responses
GET	/applications?query=video	-	200 OK <applications> <application> <name>VideoTranscoder</name> <description> Transcode mobile phone videos into other formats. </description> </application> <application> … </application> </applications> 500 ERROR

Execute an application

This application will run on machines that satisfy the constraints condition. The ones that also satisfy the preferences attributes will be given higher priority in usage. The grid middleware will create parallel tasks, each of them running in a different node (if forceDifferentNodes is false, two or more tasks can be run by one node). Later, after execution is finished, the user will receive the specified outputs.

HTTP Method	URI	Request Content Sample	Responses
POST	/executions	<execution> <applicationId> {application_id} </applicationId> <constraints> freeRAM >= 1024 </constraints> <preferences> freeCPU >= 30	201 CREATED <execution link="/ execution/{id}"/> 500 ERROR

		</preferences> <applicationType> Bag of tasks </applicationType> <arguments> --size 480x800 --format xvid </arguments> <numberOfTasks> 8 </numberOfTasks> <forceDifferentNodes> True </forceDifferentNodes> <storeStdout> False </storeStdout> <storeStderr> True </storeStderr> <storeOutputFiles> <outputDirectory> xvidVideos </outputDirectory> </storeOutputFiles> *(binaries of input files attached)* </application>	

Cancel an execution

HTTP Method	URI	Request Content Sample	Responses
DELETE	/execution/{id}	-	200 OK 500 ERROR

Get existing execution details

All the information given in the execution creation will be returned, except binaries and there is also the status tag to inform if execution was finished, is still running, waiting for available machines, etc. In case it is finished, the output is returned.

HTTP Method	URI	Request Content	Responses
GET	/execution/{id}	-	200 OK <execution> <applicationId> … <constraints> … <status> Finished </status> </execution> 404 NOT FOUND 500 ERROR

List existing executions

Existing executions can be retrieved in two different ways: one by listing executions of a specific application and another by listing all executions (if "applicationId" is not specified).

HTTP Method	URI	Request Content	Responses

GET	/executions?applicationId={applicationId}	-	200 OK <executions> <application id=”{appId}”> <execution id=”{id}”> <status> Finished </status> </execution> </application> </executions> 404 NOT FOUND 500 ERROR

Getting execution results

HTTP Method	URI	Request Content	Responses
GET	/execution/{id}/result	-	200 OK <execution> (binary output attached) </execution> 404 NOT FOUND 500 ERROR

Get NameNode configuration

This operation is Hadoop specific and does not apply to the InteGrade API.

HTTP Method	URI	Request Content	Responses
GET	/applications/{application_id}/namenode	-	200 OK <configuration>

			<property> <name>fs.default.name</name> <value>hdfs://localhost:54310/</value> </property> <property> <name>mapred.job.tracker</name> <value>localhost:54311</value> </property> </configuration> 404 NOT FOUND 500 ERROR